Implementación de un *cluster* de dispositivos Raspberry Pi para procesamiento distribuido

Daniel Estuardo Mundo Drummond

UVG UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Implementación de un *cluster* de dispositivos Raspberry Pi para procesamiento distribuido

Trabajo de graduación presentado por Daniel Estuardo Mundo Drummond para optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

2024

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Implementación de un *cluster* de dispositivos Raspberry Pi para procesamiento distribuido

Trabajo de graduación presentado por Daniel Estuardo Mundo Drummond para optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

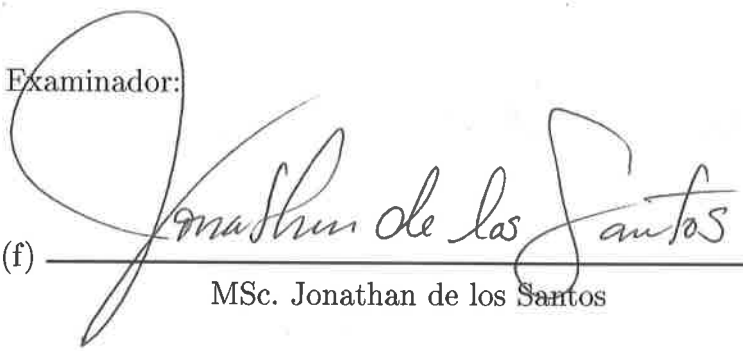
2024

Vo.Bo.:

(f) _____

MSc. Jonathan de los Santos

Tribunal Examinador:



(f) _____

MSc. Jonathan de los Santos

(f) _____

Dr. Luis Alberto Rivera

(f) _____

MSC. Diego Alberto Morales

Fecha de aprobación: Guatemala, 6 de enero de 2024.

Índice

Lista de figuras

Lista de cuadros

Resumen

En este proyecto, se aborda el concepto de procesamiento distribuido, que se refiere a una estrategia de computación en la que una tarea o proceso computacional se divide en procesos más pequeños y que se ejecutan en múltiples nodos de manera simultánea e interconectada. Teniendo en cuenta lo anterior esta investigación tiene como objetivo el desarrollo de una plataforma para evaluar el rendimiento del procesamiento distribuido en un *cluster* de Raspberry Pi 3B utilizando los protocolos I2C, UART y Ethernet, además, de comparar el rendimiento de una única Raspberry Pi, con y sin multiprocesamiento.

Para logar la comunicación entre nodos empleando el protocolo I2C y UART, se diseñaron programas en lenguaje Python, que emplean un algoritmo el cual chequea la integridad de los datos transmitidos. En el caso del protocolo Ethernet, se utilizó Slurm como el coordinador de tareas y se implementó una automatización para la configuración de nodos esclavos. Para lograr esta, se desarrolló un código en lenguaje Python que permitió establecer todas las configuraciones necesarias en cada nodo esclavo.

En cuanto a la comparación de rendimiento, se realizó una evaluación centrada en el tiempo de ejecución, analizando matrices cuadradas de dimensiones que varían desde 2×2 hasta 7×7 con múltiples pruebas para garantizar resultados robustos. Los resultados mostraron que el multiprocesamiento no mejora el rendimiento cuando se trabajan con las dimensiones antes mencionadas. También se realizaron pruebas tanto en el *cluster* de Raspberry Pi (protocolo Ethernet) y la Raspberry Pi individual (con y sin multiprocesamiento) con matrices cuadradas cuyas dimensiones son 100×100, 500×500 y 1000×1000. Los resultados obtenidos evidenciaron que la implementación de un *cluster* de Raspberry Pi mejora el rendimiento en comparación con una Raspberry Pi individual, con y sin multiprocesamiento, conforme se incrementan las dimensiones de la matriz.

En conjunto, este proyecto representa un enfoque sólido en la investigación de procesamiento distribuido en dispositivos Raspberry Pi, con énfasis en la eficiencia, la automatización y la optimización del rendimiento de la red. Los resultados obtenidos respaldan la viabilidad y utilidad de esta estrategia en entornos de *cluster* de Raspberry Pi, abriendo nuevas posibilidades en el campo de la computación distribuida.

Abstract

In this project, the concept of distributed processing is addressed, which refers to a computing strategy in which a computational task or process is divided into smaller processes that are executed on multiple nodes simultaneously and interconnected. Taking into account the above, this research aims to develop a platform to evaluate the performance of distributed processing in a Raspberry Pi 3B *cluster* using the I2C, UART and Ethernet protocols, in addition to comparing the performance of a unique Raspberry Pi, with and without multiprocessing.

To achieve communication between nodes using the I2C and UART protocol, programs were designed in Python language, which use an algorithm which checks the integrity of the transmitted data. In the case of the Ethernet protocol, Slurm was used as the task coordinator and automation was implemented for the configuration of slave nodes. To achieve this, a code was developed in Python language that allowed all the necessary configurations to be established on each slave node.

Regarding the performance comparison, an evaluation focused on the execution time was performed, analyzing square matrices of dimensions ranging from 2×2 to 7×7 with multiple tests to ensure robust results. The results showed that multiprocessing does not improve performance when working with the aforementioned dimensions. Tests were also performed on both the Raspberry Pi *cluster* (Ethernet protocol) and the individual Raspberry Pi (with and without multiprocessing) with square matrices whose dimensions are 100×100, 500×500 and 1000×1000. The results obtained showed that the implementation of a Raspberry Pi *cluster* improves performance compared to an individual Raspberry Pi, with and without multiprocessing, as the dimensions of the matrix increase.

Together, this project represents a strong focus on distributed processing research on Raspberry Pi devices, with an emphasis on efficiency, automation, and network performance optimization. The results obtained support the viability and usefulness of this strategy in Raspberry Pi *cluster* environments, opening new possibilities in the field of distributed computing.

CAPÍTULO 1

Introducción

En el ámbito de la informática orientada a mejorar el rendimiento y la eficiencia, los *cluster* de Raspberry Pi han surgido como una opción atractiva para el procesamiento distribuido. Este estudio se enfoca en crear una plataforma que evalúe el rendimiento de estos *cluster* en comparación con Raspberry Pi individuales, tanto en configuraciones estándar como con multiprocesamiento.

Para lograrlo, se desarrolló un programa que permitirá comparar el rendimiento de los *cluster* y las Raspberry Pi con y sin multiprocesamiento. Además, se analizó una variedad de protocolos de comunicación utilizados para comunicar los nodos del *cluster*, lo cual es esencial para optimizar el rendimiento distribuido. A través de comparativas exhaustivas, se buscó medir el impacto real del procesamiento distribuido en términos de velocidad, eficiencia y capacidad de carga.

Este trabajo también tiene como objetivo facilitar futuras investigaciones en este campo. Con este propósito, se generó documentación detallada y de fácil acceso, proporcionando a otros investigadores y entusiastas las herramientas necesarias para aprovechar estos hallazgos en proyectos y desarrollos por venir.

CAPÍTULO 2

Antecedentes

2.1. Aplicaciones de la Raspberry Pi en la Universidad

En la Universidad del Valle la computadora de mono placa Raspberry Pi ha sido implementada para distintas aplicaciones. Una de estas aplicaciones es con fines didácticos, como lo son los temas impartidos en el curso de Digital 3 el cual tiene el propósito de combinar distintas aplicaciones computacionales a un entorno de sistemas embebidos. Entre los temas que se tratan en dicho curso se puede destacar *multithreading*, *multiprocessing*, *task scheduling*, *pipes* y distintas aplicaciones de IoT. Otra de las aplicaciones de la Raspberry Pi en la Universidad, es su implementación como computadora central para el proyecto del Robot Explorador Modular (ver Figura 1). En dicho proyecto la Raspberry Pi se encarga tanto de recopilar la información de los sensores, como de gestionar la plataforma *Robot Operating System* [1].

Figura 1: Fotografía del Robot Explorador Lunar sobre plataforma de pruebas [1].

2.2. Proyectos con *cluster* Raspberry Pi

2.2.1. *A Modular Based Design for Distributed Computing Systems*

Es un proyecto desarrollado por los estudiantes David Kruse, Casey Bedford, Brandon Jones y Dallas Fletchall, pertenecientes al grupo Capstone, de la Universidad de Missouri en el cual se construye un sistema de cómputo distribuido utilizando dispositivos Raspberry Pi. En el documento [2] se describe tanto el alcance del proyecto como los logros y limitaciones del mismo, además de la distribución de las tareas para cada uno de los integrantes del grupo. El proyecto se implementó como un *cluster* con 28 Raspberry Pi (o nodos) en total, que a su vez se dividían en 2 *fundamental array* (FA), de los cuales uno era un arreglo de 15 nodos y el otro un arreglo de 13 nodos. Todos estos nodos se conectaban entre si por medio de Ethernet (ver Figura 2).

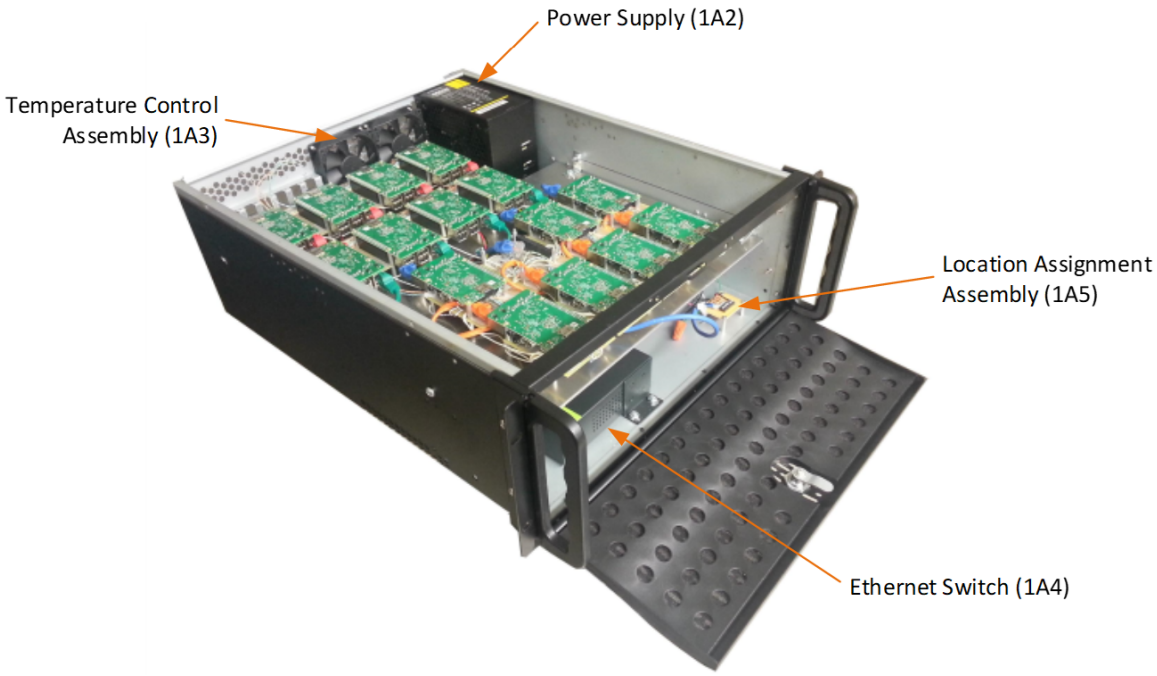


Figura 2: Fotografía del *cluster* [2].

El integrante David Kruse fue el encargado del diseño de un circuito de placa impresa (PCB, por sus siglas en inglés) el cual incluye la conexión de la Raspberry Pi (nodo) con la alimentación, un circuito para el reinicio del nodo y de un *driver* para determinar la ubicación del nodo dentro del *cluster* (ver Figura 3) [3].

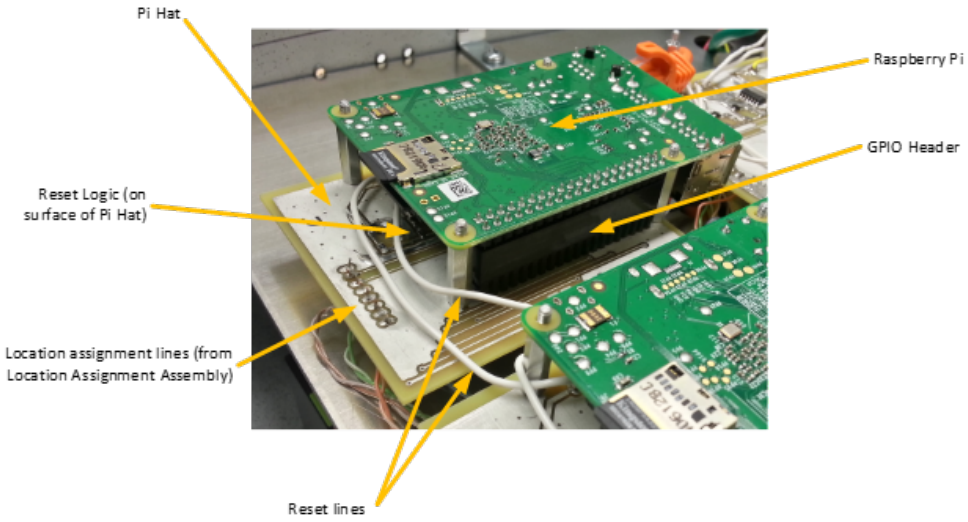


Figura 3: Fotografía del ensamblaje de un nodo del *cluster* [3].

Dallas Fletchall contribuyó ensamblando el proyecto y, diseñando una aplicación para el envío y recepción de los mensajes provenientes del servidor [4]. Casey Bedford se encargó del diseño de una aplicación para una computadora externa con el fin monitorear y controlar el *cluster*, en el documento [5] se explica con mayor detalle el alcance de la aplicación.

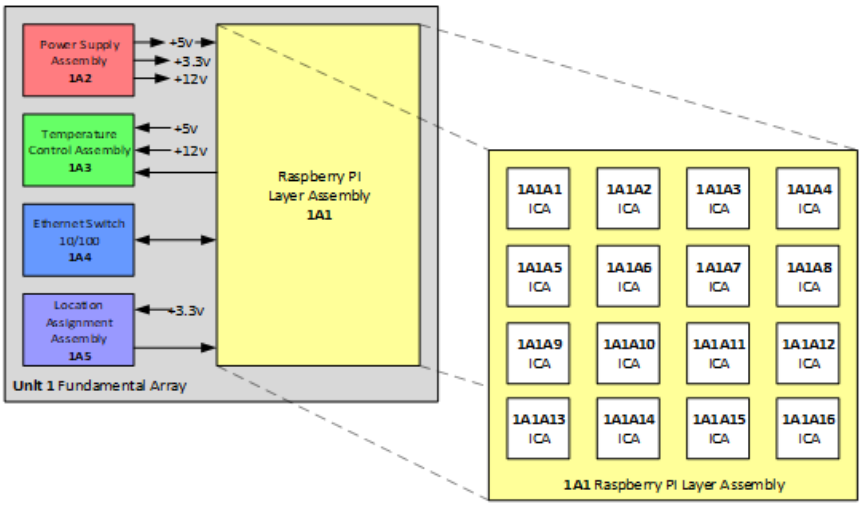


Figura 4: Diagrama de bloques del funcionamiento [2].

En el documento [6] se evidencia la contribución de Brando Jones, la cual consistió en desarrollar e instalar el *software* necesario para hacer al *cluster* funcional, crear los *drivers* para realizar los reportes de temperatura y reinicio de la Raspberry pi, además de elaborar el programa para evaluar el funcionamiento del *cluster*. La distribución de procesos se realizó desde el nodo maestro, ubicado en la posición 1A1A (ver Figura 4), por medio de la librería del software MPI (software para aplicaciones de procesamiento paralelo). Para evaluar el funcionamiento del *cluster* el programa desarrollado realizaba una multiplicación de matrices de $N \times M$ por $M \times O$.

2.2.2. *NPi-Cluster: A Low Power Energy-Proportional Computing Cluster Architecture*

En este artículo se presenta un *cluster* de Raspberry Pi (o como le denominan NPi-Cluster) en la cual sus nodos automáticamente se encienden o apagan dependiendo de a la demanda de procesamiento (ver Figura 5). Como parte de sus pruebas se construyó un *cluster* de 6 nodos y un maestro que tenía la tarea de realizar distintas solicitudes de HTTP. Las conexiones de cada uno de los nodos se realizó por medio de Ethernet [7].

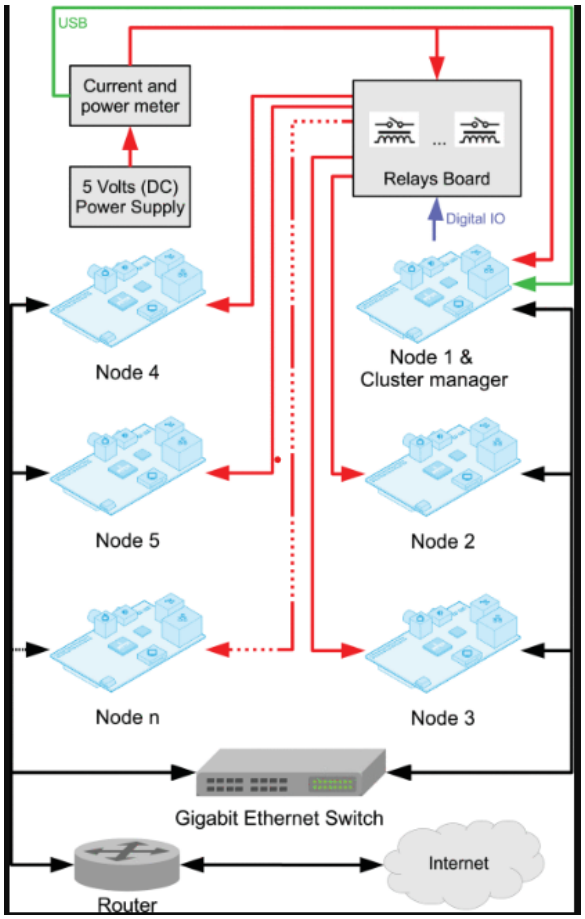


Figura 5: Diagrama del *cluster* diseñado [7].

CAPÍTULO 3

Justificación

Hoy en día, los *cluster* poseen una gran popularidad en varios campos, entre los cuales se incluye la informática de alto rendimiento, los centros de datos, la investigación científica y los sistemas distribuidos. La razón de esto es, que un *cluster* combina la potencia informática y los recursos de varias máquinas, por lo cual estos ofrecen no solo un mejor rendimiento, sino que también escalabilidad, tolerancia a fallas y la capacidad de manejar cargas de trabajo complejas. Debido a su asequibilidad, versatilidad y facilidad de configuración los *cluster* de dispositivos Raspberry Pi son una de las opciones de *cluster* más populares hoy en día.

Una característica fundamental de los dispositivos Raspberry Pi es su capacidad para proporcionar al usuario pines de propósito general. Esta característica abre la puerta a la exploración de alternativas a la comunicación a través del protocolo Ethernet. Al implementar protocolos propios de microcontroladores, como I2C o UART, no solo se elimina la necesidad de *hardware* adicional, sino que también simplifica significativamente la infraestructura del *cluster*. Otra ventaja de la implementación de estos protocolos es la reducción sustancial de los costos de implementación y del consumo de energía. Considerando estas ventajas, se plantea la posibilidad de aprovechar los recursos disponibles en los dispositivos Raspberry Pi para mejorar la eficiencia y la accesibilidad de los *clusters*.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Elaboración de una plataforma para comprobar el rendimiento de procesamiento distribuido para *cluster* de Raspberry Pi con distintos protocolos de comunicación y, comparar el rendimiento de los mismos contra una única Raspberry Pi con y sin multiprocesamiento.

4.2. Objetivos específicos

- Elaborar un programa para comparar el rendimiento de los *cluster* y la Raspberry Pi con y sin multiprocesamiento.
- Evaluar distintos protocolos de comunicación para la conexión entre los nodos del *cluster*.
- Evaluar el rendimiento de los *cluster* contra la Raspberry Pi con y sin multiprocesamiento.
- Elaborar documentación para facilitar la implementación del trabajo para proyectos futuros.

CAPÍTULO 5

Alcance

El alcance del presente trabajo se enfocó en la creación y desarrollo de códigos en lenguaje Python para la configuración y el procesamiento distribuido en un *cluster* de Raspberry Pi. El objetivo principal fue el desarrollar una serie códigos los cuales permiten realizar procesamiento distribuido utilizando los protocolos I2C y UART, al ejecutar sus versiones de códigos correspondientes. Estos códigos se probaron con éxito en el *cluster* de Raspberry Pi, demostrando que es posible realizar procesamiento distribuido utilizando los protocolos I2C y UART.

También, como parte de este trabajo se realizaron pruebas de rendimiento para determinar el protocolo de comunicación más eficiente para el procesamiento distribuido en un *cluster* de Raspberry Pi. Estas pruebas de rendimiento se realizaron utilizando los protocolos I2C, UART y Ethernet, con Ethernet demostrando un rendimiento superior de los tres. Y, por último, se realizó una comparación de rendimiento entre una Raspberry Pi con y sin multiprocesamiento para determinar si el multiprocesamiento mejora el rendimiento de la Raspberry Pi, los resultados obtenidos demostraron que el multiprocesamiento no mejora el rendimiento de la Raspberry Pi, cuando se trabajan con dimensiones de matrices muy pequeños.

Además, como parte de este trabajo se desarrolló un código en lenguaje Python el cual permite realizar la configuración de los nodos esclavos, de manera automática, de un *cluster* de Raspberry Pi, que emplea el protocolo Ethernet para la comunicación de sus nodos. Este código se ejecuta cuando un cliente de DHCP realiza su primera solicitud al servidor que se encuentra operando en el puerto Ethernet del nodo maestro. Al momento de ejecutarse el código, éste realiza la instalación de paquetes y dependencias requeridas, la configuración de parámetros de red y comunicación, así como la preparación de los nodos para la participación en trabajos y tareas distribuidas.

CAPÍTULO 6

Marco teórico

6.1. Raspberry Pi 3 modelo B

La Raspberry Pi 3 modelo B es una computadora de placa única desarrollada por la Fundación Raspberry Pi. Salió al mercado en febrero del 2016 para reemplazar al dispositivo Raspberry Pi 2 modelo B. En el Cuadro 1 se puede visualizar las características de ésta y en la Figura 6 es posible apreciar el *pinout* [8].

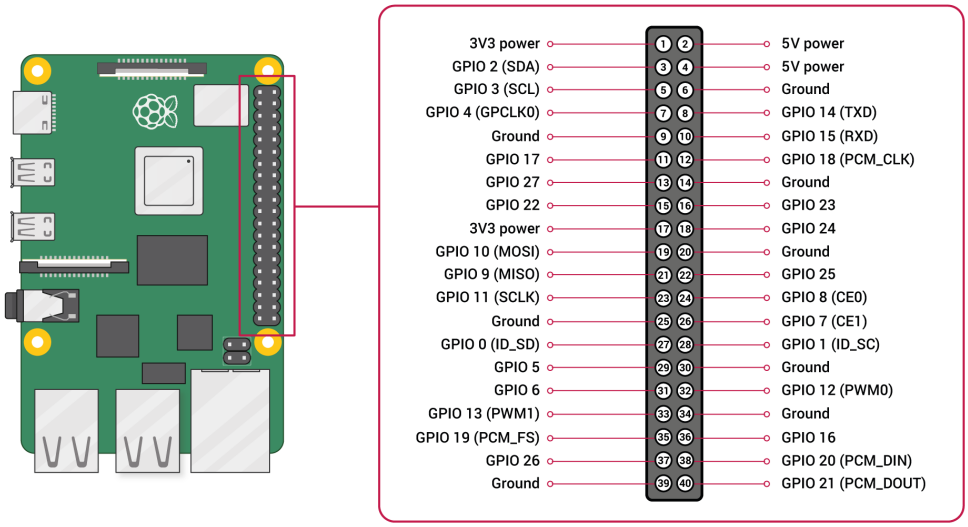


Figura 6: Fotografía del *pinout* de la Raspberry Pi 3 B [9].

Características	Raspberry Pi 3 B
Procesador	Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
Frecuencia de operación	1.2GHz
Unidad de procesamiento gráfico	Dual Core VideoCore IV
Memoria Flash	32GB Micro SD
Memoria RAM	1GB
Puertos USB	4
Puertos Ethernet	1
Velocidad máxima del puerto Ethernet	100 Mbp/s
Voltaje máximo de pines de entrada/salida	3.3V
Módulos UART	1
Módulos I2C	2 (1 exclusivo para EEPROM)

Cuadro 1: Características de la Raspberry Pi 3 B.

6.2. Protocolo I2C

I2C (Inter-Integrated Circuit) es un protocolo de comunicación serial síncrono ampliamente utilizado, desarrollado por Philips Semiconductors (ahora NXP Semiconductors) a principios de la década de 1980. Proporciona un método simple y eficiente para interconectar múltiples circuitos integrados en un solo bus.

6.2.1. Interfaz

El protocolo I2C consiste en 2 cables bidireccionales para la comunicación entre el maestro y el esclavo, siendo estos:

- **Reloj**: Conocido como SCL (*serial clock line*), en este cable se transmite la señal de reloj generada desde el dispositivo maestro. En la Raspberry Pi 3 B el pin para este cable es el GPIO 3 o pin 5 en físico.
- **Bus de datos**: Conocido como SDA (*serial data line*), a través de este cable se transmite la información proveniente de los distintos dispositivos, tanto maestro como esclavo. En la Raspberry Pi 3 B el pin para este cable es el GPIO 2 o pin 3 en físico.

6.2.2. Transmisión de la información

Para iniciar la comunicación I2C, el dispositivo maestro debe enviar la señal de reloj y realizar una transición de 1 lógico a 0 lógico mientras la señal de reloj se encuentre en su periodo de 1 lógico (ver Figura 7), posterior a ello el dispositivo maestro debe de escribir en el bus de datos la dirección del dispositivo esclavo, con el que se desee comunicar (ver Figura 7). Para finalizar la comunicación el dispositivo debe de hacer una transición de 0 lógico a 1 lógico mientras la señal de reloj se encuentre en su periodo de 1 lógico (ver Figura 7).

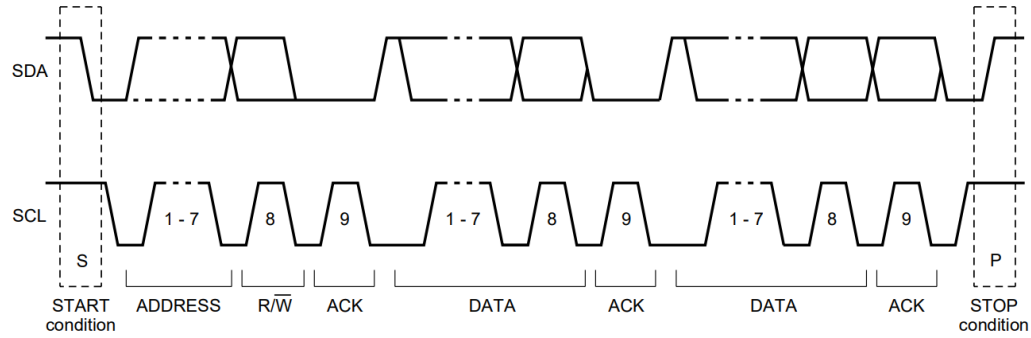


Figura 7: Diagrama de tiempo del protocolo I2C [10].

Cuando el bus de datos se encuentra libre, tanto SCL como SDA deben de encontrarse en un nivel lógico de 1 (o en alto). Los niveles lógicos para este protocolo se encuentran asociados a la alimentación del dispositivo, siendo común 30 % de la alimentación para considerarlo como nivel lógico de 0 (o bajo) y 70 % de la alimentación como nivel lógico de 1. El protocolo considera únicamente válido el cambio de estado (de alto a bajo o viceversa) en los periodos en los cuales la señal de SCL este en bajo, es decir durante los periodos altos de SCL la señal de SDA debe ser estable (ver Figura 8).

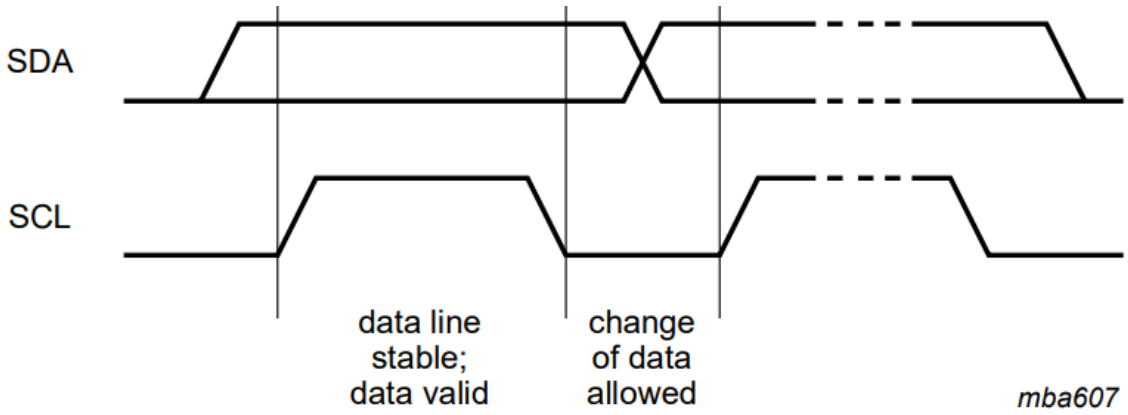


Figura 8: Diagrama de tiempo para la transferencia de datos [10].

El protocolo I2C permite la conexión de varios dispositivos maestro, generando esto, el caso de que varios dispositivos deseen iniciar la comunicación provocando así colisiones. Para evitar esto último, el protocolo realiza un proceso de "arbitraje" para seleccionar quien transmite primero la información. Dicho proceso de arbitraje consiste en que los dispositivos maestros comparan la información, bit por bit, en el bus de datos (SDA) con la información que debió de transmitirse en el momento que la señal reloj (SCL) se encuentre en alto. El proceso continúa hasta que alguno de los dispositivos maestros detecte que hay alguna diferencia en lo leído en el bus de datos y lo que debe de transmitir, en ese momento el dispositivo maestro que lo detectó procede apagar su *driver* para el SDA (ver Figura 9).

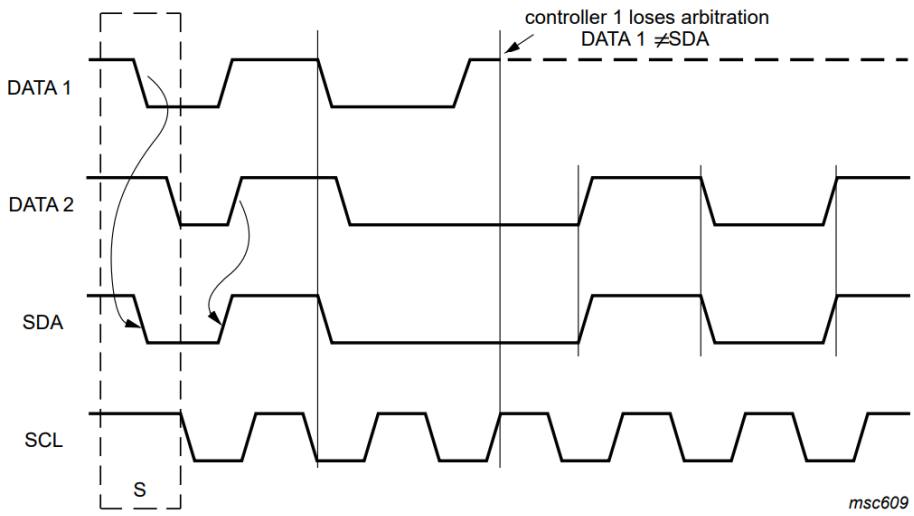


Figura 9: Diagrama de tiempo para proceso de arbitraje [10].

El protocolo I2C transmite la información a través de bytes de los cuales el receptor provee de un ACK (*Acknowledge*), para la confirmación de la recepción de cada byte. La transmisión de la información ocurre enviando el bit más significativo (MSB, por sus siglas en inglés) de primero, ver Figura 10. El primer byte que se envía al comienzo de la comunicación incluye la dirección del dispositivo con el que se desea comunicar, siendo esta de 7 bits, y de la acción que se desea realizar, escribir o leer. En la Figura 11 se puede observar el formato antes mencionado.

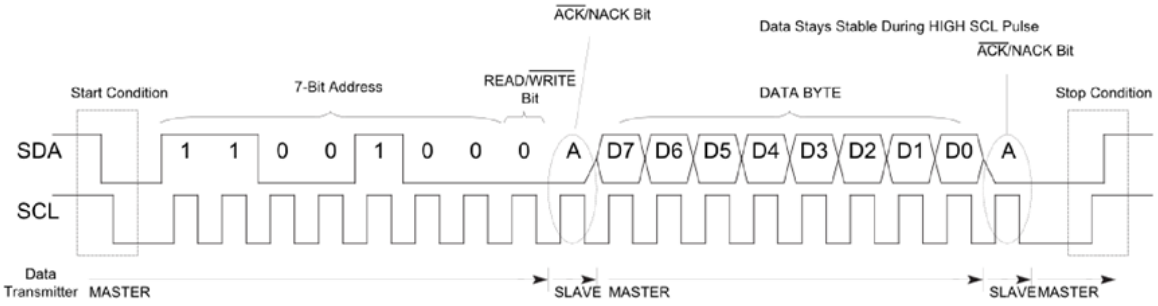


Figura 10: Ejemplo de comunicación correcta I2C [11].

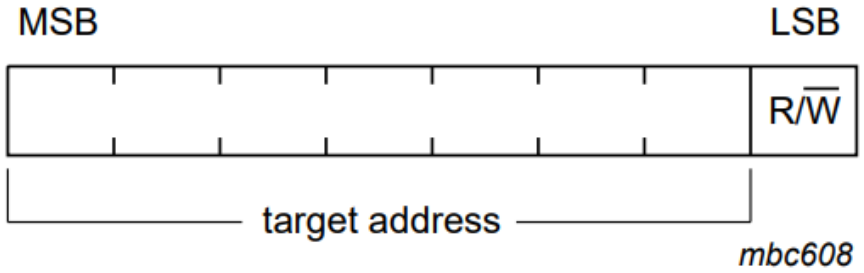


Figura 11: Formato del primer byte posterior al inicio de la comunicación I2C [10].

6.3. Protocolo UART

El Protocolo UART (Universal Asynchronous Receiver/Transmitter) es un estándar de comunicación serial ampliamente utilizado en electrónica y sistemas embebidos. A diferencia de otros protocolos (como I2C) UART es asíncrono, es decir, que el protocolo no requiere un reloj común entre los dispositivos conectados, lo que lo hace altamente versátil y adecuado para una variedad de aplicaciones [12].

6.3.1. Interfaz

El protocolo UART consiste en 2 cables unidireccionales para la comunicación entre los dispositivos, siendo estos:

- **Transmisor**: Conocido como TX (*Transmitter*), en este cable se transmite la información desde el dispositivo 1 hacia el dispositivo 2 (ver Figura 12). En la Raspberry Pi 3 B el pin para este cable es el GPIO 14 o pin 8 en físico.
- **Receptor**: Conocido como RX (*Receiver*), en este cable se recibe la información transmitida desde el dispositivo 2 hacia el dispositivo 1 (ver Figura 12). En la Raspberry Pi 3 B el pin para este cable es el GPIO 15 o pin 10 en físico.

Figura 12: Diagrama de conexión del protocolo UART [12].

6.3.2. Transmisión de la información

UART utiliza una transmisión de datos en serie, donde los bits se transmiten de manera secuencial, comenzando con el bit de inicio, seguidos de los bits de datos, bits de paridad (si se utilizan) y bits de parada. La velocidad de transmisión se mide en baudios (baud rate) y determina la cantidad de bits transmitidos por segundo. Ambos dispositivos en la comunicación deben configurarse a la misma velocidad de baudios para una comunicación exitosa.

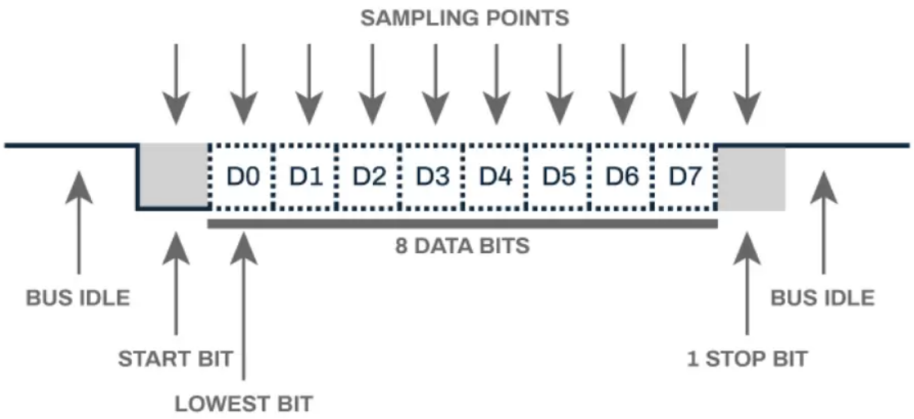


Figura 13: Diagrama de tiempo del protocolo UART [12].

Para iniciar la comunicación, el dispositivo 1 debe de enviar un bit de inicio, el cual es un 0 lógico, seguido de los datos que se desean transmitir, y finalmente un bit de parada, el cual es un 1 lógico (ver Figura 13). El dispositivo 2 debe de leer el bit de inicio y esperar un tiempo igual a la mitad del periodo de un bit para leer el primer bit de datos, posterior a ello debe de leer los bits de datos cada periodo de bit hasta que se lea el bit de parada (ver Figura 13).

6.4. Protocolo Ethernet

Ethernet es un protocolo de red ampliamente utilizado que proporciona un método estándar para conectar dispositivos dentro de una red de área local (LAN, por sus siglas en inglés). Fue desarrollado en la década de 1970 por el Centro de Investigación de Palo Alto (PARC) de Xerox Corporation y desde entonces ha evolucionado para convertirse en el estándar de facto para las conexiones LAN cableadas [13].

Ethernet como protocolo opera en dos capas del modelo OSI (*Open System Interconnection*) siendo esta la física y la de enlace, ver Figura 14. Define las características eléctricas, mecánicas y funcionales del medio físico a través del cual se transmite datos [14]. Originalmente, Ethernet utilizaba cables coaxiales, pero hoy en día comúnmente utiliza cables de par trenzado (como Cat5e o Cat6) o cables de fibra óptica [15].

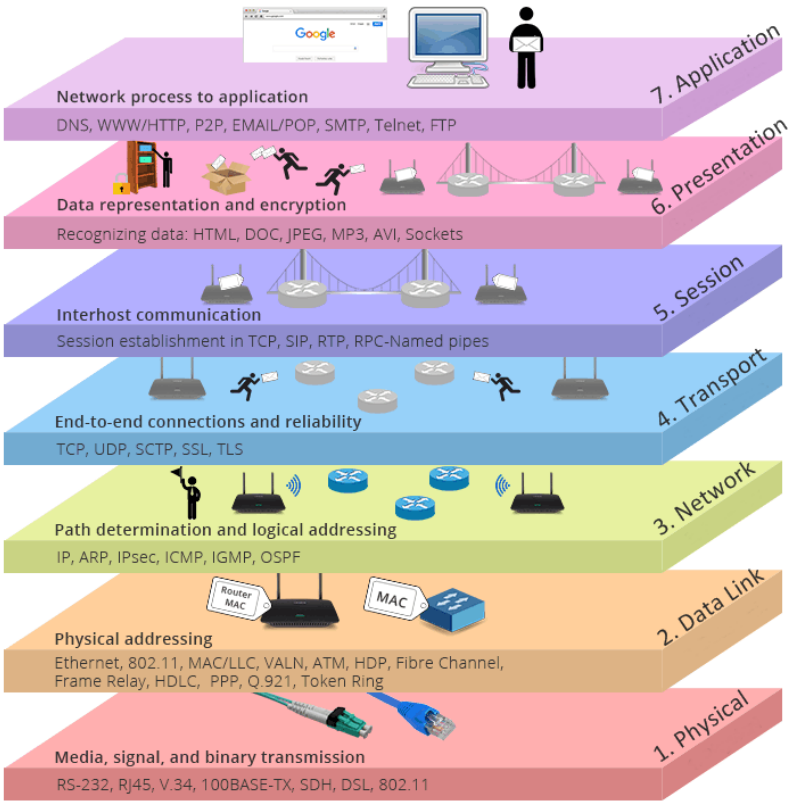


Figura 14: Estructura del modelo OSI [16].

6.4.1. Estructura

En contexto de la capa 2 del modelo OSI la información recibida se conoce como trama (*frame* en inglés), ver Figura 15. En dicho trama es posible observar lo siguiente:

- **Campo de información**
  En este campo se encuentra el datagrama (análogo de trama en la capa 3). Este campo tiene como mínimo 46 bytes de información y como máximo 1500 bytes ya que, esta definido en el IEEE 802.3, original, que la unidad máxima de transmisión (MTU, por sus siglas en ingles) del Ethernet es de 1500 bytes [13]. Existen otras versiones donde el MTU puede llegar hasta los 9000 bytes, sin embargo, este no se encuentra estandarizado por parte de la IEEE [17].

- **Campo de dirección de destino**
  Este campo contiene la dirección MAC (*media access control*) del adaptador destino, dicha dirección tiene un tamaño de 6 bytes y es un identificador físico único. Si la dirección MAC recibida coincide con el adaptador, el dispositivo procede a mandar el datagrama a la capa 3 en caso contrario esté deshecha el paquete.

- **Campo de dirección de origen**
  Este campo contiene la dirección MAC del adaptador de red que transmitió la trama, este campo tiene un tamaño de 6 bytes.

- **Campo de tipo**
  Este campo permite al protocolo Ethernet saber a qué protocolo de capa de red pertenece el datagrama.
- **Campo de *Cyclic redundancy check***
  El campo del CRC permite a al protocolo Ethernet detectar errores en la trama.
- **Campo de *Preambble***
  Este campo permite a al protocolo Ethernet sincronizar el reloj del receptor al del transmisor.

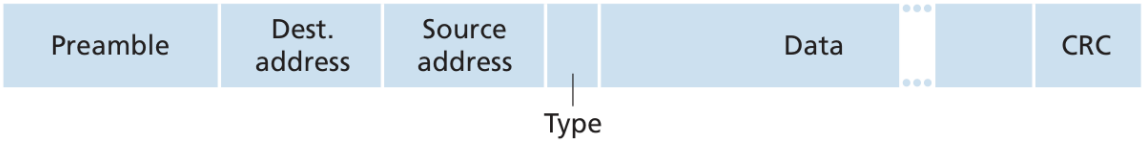


Figura 15: Estructura de la trama [13].

La estructura del trama puede sufrir cambios cuando se configuran VLANs (*virtual local area networks*), las cuales tienen como objetivo principal separar el tráfico de capa 2 para así tener un mejor control de las redes, además, de mayor privacidad. Las modificaciones que sufre la trama son observables en la Figura 16 a continuación se describe cada uno de los nuevos campos:

- **Etiqueta del identificador de protocolo**
  Este campo tiene un tamaño de 2 bytes y éste tiene un valor, en hexadecimal, de $81 - 00$.
- **Etiqueta de control de tráfico**
  Este campo tiene un tamaño de 2 bytes, en los cuales 12 bits corresponde al identificador de la VLAN y 3 bits los cuales sirven para definir la prioridad que tiene la trama.

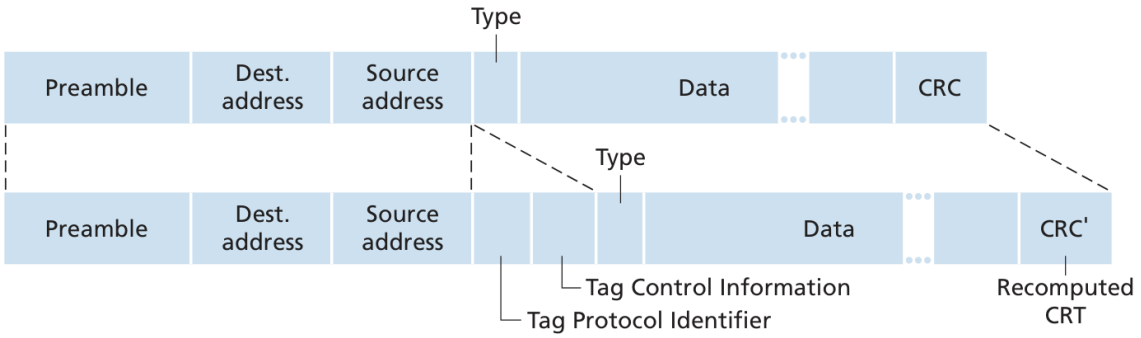


Figura 16: Estructura de la trama con VLAN [13].

6.4.2. Estándares del Ethernet

Los estándares de Ethernet están definidos por el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, por sus siglas en inglés) [18]. Los estándares más ampliamente adoptados incluyen Ethernet (IEEE 802.3), Fast Ethernet (IEEE 802.3u) y Gigabit Ethernet (IEEE 802.3ab). Estos estándares especifican las características físicas, las tasas de datos y los métodos de señalización para las redes Ethernet [13].

6.4.3. Relación Ethernet y la Capa 3

El Ethernet (Capa 2) y la Capa 3 (o protocolo de red) son componentes esenciales en una red de computadoras. En la Capa 2 los datos se encapsulan en tramas que contienen las direcciones MAC de los dispositivos con el fin de identificarlos en la misma red local. Esto debido a que la dirección MAC se utiliza para la comunicación dentro de una red local [13]. No obstante, cuando los datos deben viajar entre diferentes redes, la Capa 3 entra en juego. Los *routers* en esta capa determinan cómo dirigir los paquetes de datos a través de múltiples dispositivos y redes [13]. En esta Capa, las direcciones IP son fundamentales para tomar decisiones de enrutamiento adecuadas.

Conceptos clave de la Capa 3

Para comprender la Capa 3, es esencial explorar conceptos clave relacionados con esta capa como lo son:

- **Enrutamiento**:
  En la Capa 3, los *router* son los dispositivos encargados de dirigir el tráfico de datos entre diferentes redes. Utilizan tablas de enrutamiento y algoritmos para tomar decisiones sobre cómo enviar los paquetes de datos hacia su destino [13].

- **Direccionamiento IP**:
  Las direcciones IP son etiquetas numéricas únicas asignadas a dispositivos en una red. Existen dos versiones de este direccionamiento:

  1. IPv4 (*Internet Protocol version 4*): Éste utiliza direcciones de 32 bits, generalmente representadas en formato decimal [13].
  2. IPv6 (*Internet Protocol version 6*): Éste utiliza direcciones de 128 bits, que se representan en formato hexadecimal [13].

  Estas direcciones permiten que los routers y otros dispositivos de red identifiquen y enruten los paquetes de datos correctamente.

- **Subredes**:
  Las subredes son divisiones más pequeñas de una red más grande. Dividir una red en subredes puede ayudar a gestionar las direcciones IP de manera más eficiente y a aislar secciones de la red para un mejor rendimiento y seguridad [13].

DHCP

Dentro del contexto de la Capa 3, es fundamental garantizar que los dispositivos cuenten con direcciones IP únicas. Por esta razón, el Protocolo de Configuración Dinámica de Host (conocido como DHCP por sus siglas en inglés) se ha convertido en un estándar de red ampliamente utilizado. Este protocolo permite la asignación automática de direcciones IP únicas y otros parámetros de configuración, como la máscara de subred, la puerta de enlace predeterminada y los servidores DNS, sin requerir que el administrador de red configure manualmente cada dispositivo [13]. El proceso DHCP sigue un ciclo conocido como DORA, que consta de las siguientes etapas:

1. **Descubrimiento**: En esta etapa, un dispositivo que se conecta a la red y necesita una dirección IP envía un mensaje de descubrimiento DHCP a la red [13]. Este mensaje se difunde a través de la red o se envía a un servidor DHCP específico si se conoce su dirección [19].
2. **Oferta**: Cuando un servidor DHCP recibe una solicitud de descubrimiento, responde con un mensaje de oferta DHCP [13]. En este mensaje, el servidor ofrece una dirección IP disponible junto con otros parámetros de configuración que el dispositivo puede aceptar [19].
3. **Solicitud**: El dispositivo que necesita una dirección IP selecciona una de las ofertas recibidas y envía un mensaje de solicitud DHCP al servidor elegido [13]. Esto informa al servidor que el dispositivo desea utilizar la dirección IP ofrecida [19].
4. **Aprobación**: Finalmente, el servidor DHCP envía un mensaje de aprobación DHCP al dispositivo, confirmando que la dirección IP solicitada se ha asignado al dispositivo [13]. El dispositivo puede ahora utilizar esta dirección IP y otros parámetros de configuración [19].

Este proceso asegura que los dispositivos obtengan direcciones IP de manera eficiente y evita conflictos de direcciones, ya que el servidor DHCP asigna direcciones que están disponibles que aún no se han asignado a otros dispositivos.

Network Address Translation

En el contexto de la Capa 3, uno de los mayores desafíos al trabajar con IPv4 es la escasez de direcciones IP públicas. Por esta razón, surge la tecnología de *Network Address Translation* (NAT) como una de las soluciones más utilizadas. NAT tiene como función principal permitir que varios dispositivos en una red local compartan una única dirección IP pública. Esto se logra mediante la traducción de las direcciones IP privadas de los dispositivos locales a una dirección IP pública cuando se comunican con redes externas [19]. El proceso de traducción de direcciones IP en NAT funciona de la siguiente manera:

1. Un dispositivo local envía un paquete de datos con su dirección IP privada como dirección de origen y la dirección IP del servidor de destino en Internet como dirección de destino [19].

2. El rúter NAT, que actúa como intermediario, modifica la dirección IP de origen del paquete, reemplazándola con su propia dirección IP pública [19].

3. El paquete se envía a través de Internet con la dirección IP pública del rúter NAT como dirección de origen [19].

4. Cuando el servidor en Internet responde, lo hace a la dirección IP pública del rúter NAT [19].

5. El rúter NAT recibe la respuesta, consulta su tabla de traducción y modifica la dirección IP de destino del paquete, reemplazándola con la dirección IP privada del dispositivo local [19].

6. Finalmente, el paquete se envía de regreso al dispositivo local [19].

Iptables

La Capa 3, encargada de las funciones de enrutamiento y gestión de paquetes a nivel de protocolo de red, guarda una estrecha relación con iptables en el contexto de sistemas Linux. Las iptables son una herramienta de *firewall* y enrutamiento de paquetes esencial en el entorno de Linux, ya que desempeñan un papel fundamental al permitir a los administradores de sistemas controlar y gestionar el flujo de tráfico de red, así como establecer políticas de seguridad y enrutamiento específicas [20]. En otras palabras, esta herramienta permite definir reglas precisas que determinan qué paquetes se permiten y cuáles se bloquean en función de diversos criterios, como direcciones IP, puertos y protocolos.

La flexibilidad y versatilidad de iptables se destacan como atributos clave, ya que permiten a los administradores configurar reglas altamente personalizadas que se adaptan a las necesidades específicas de la red. Esto no se limita únicamente a la capacidad de filtrar y redirigir paquetes en función de la Capa 3, sino que también abarca la aplicación de reglas avanzadas de seguridad y enrutamiento según sea necesario. Uno de sus usos más comunes es la implementación de *Network Address Translation* (NAT) en dispositivos Linux [20].

6.5. Procesamiento distribuido

El procesamiento distribuido se refiere a la división de tareas de cómputo entre múltiples dispositivos conectados en una red, lo cual permite ejecuciones en paralelo y un mejor rendimiento. Los modelos de procesamiento centralizados tradicionales enfrentan limitaciones en el manejo de tareas de gran escala e intensivas en cómputo de manera eficiente. El procesamiento distribuido ofrece varias ventajas, incluyendo escalabilidad, tolerancia a fallos, equilibrio de carga y mayor poder de cómputo [2].

La investigación del procesamiento distribuido ha abarcado varias arquitecturas,protocolos de comunicación, estrategias de asignación de tareas y modelos de programación. Los entornos prominentes como Apache Hadoop, *Message Passing Interface* (MPI) y MapReduce se han utilizado ampliamente para el procesamiento distribuido en sistemas de gran escala.

Estos entornos proporcionan abstracciones y herramientas para gestionar la distribución, coordinación y sincronización de tareas en múltiples nodos [21].

6.6. *Message Passing Interface*

Es tanto un protocolo de comunicación como un modelo de programación ampliamente utilizado para implementar aplicaciones de cómputo paralelo y distribuido. MPI fue diseñado para facilitar el intercambio de datos y mensajes entre los procesos, permitiéndoles trabajar de manera colaborativa en una tarea compartida. La primera versión del protocolo se publicó en el 5 de mayo de 1994 con el nombre de MPI-1.0 y su versión más actual es la MPI-4.0 publicada en el 9 de julio del 2021 [21].

Admite una variedad de opciones de comunicación, entre las cuales está la comunicación punto a punto (envío y recepción de mensajes entre procesos específicos) y la comunicación colectiva (comunicación que involucra a un grupo de procesos). Además, ofrece diversas características y funcionalidades, incluyendo soporte para diferentes tipos de datos, comunicación no bloqueante, gestión de procesos, creación dinámica de procesos y tolerancia a fallos [21].

Las implementaciones de MPI están disponibles para una amplia gama de sistemas, incluyendo *clusters*, supercomputadoras y arquitecturas de memoria distribuida. Se ha convertido en el estándar de facto para el cómputo paralelo y distribuido, y se utiliza ampliamente en simulaciones científicas, cálculos numéricos, análisis de datos y otras aplicaciones de cómputo de alto rendimiento [21].

Si bien MPI ofrece beneficios significativos para el cómputo distribuido y paralelo, también presenta desafíos, como la gestión de la sobrecarga de comunicación, el equilibrio de carga y la minimización de los retrasos de sincronización. Un diseño adecuado y la optimización de las aplicaciones basadas en MPI son cruciales para lograr un alto rendimiento y escalabilidad [21].

6.6.1. Operación MPI

Una operación MPI es una secuencia de pasos que debe de seguirse para lograr establecer y habilitar la sincronización y/o transferencia datos [21]. En general, en operaciones MPI es posible observar los siguientes escenarios:

- **Inicialización**: En este escenario el MPI se inicializa y se le asigna a cada proceso un identificador o rango único.
- **Distribución de datos**: El MPI en este escenario procede a dividir los datos de entrada entre cada uno de los procesos.

- **Cómputo**: En este escenario se lleva a cabo la realización de las tareas asignadas para cada uno de los procesos.
- **Comunicación**: El escenario surge cuando los procesos requieren de intercambiar y/o sincronizar su progreso.
- **Sincronización**: El escenario ocurre cuando los procesos necesitan de sincronizar sus ejecuciones para asegurar un resultado correcto.
- **Agregación**: En este escenario, los procesos combinan sus resultados individuales para obtener el resultado final.
- **Finalización**: Este escenario ocurre únicamente cuando se completan todos los cálculos, en este se finalizan los procesos y el protocolo MPI.

6.7. Slurm

Es un sistema de gestión de *clusters* y programación de trabajos de código abierto. Es escalable y tolerante a fallos, diseñado para *clusters* Linux [22]. Slurm opera sin necesidad de modificar el *kernel* y funciona de manera autónoma. Cumple tres funciones principales como administrador de cargas de trabajo:

1. Asignar a los usuarios el acceso a los recursos [22].
2. Proporcionar un entorno para iniciar, ejecutar y monitorear trabajos en nodos asignados [22].
3. Gestionar la competencia por recursos mediante una cola de trabajo [22].

Slurm también posee complementos para contabilidad, reserva avanzada, programación en grupo, programación de relleno posterior, selección optimizada de recursos, límites de recursos por usuario y priorización de trabajos multifactoriales [22].

6.7.1. Arquitectura

Slurm posee una arquitectura modular (ver Figura 17), la cual consta de varios componentes que gestionan de manera conjunta todo el proceso para completar los trabajos. Entre estos componentes se incluyen:

1. **Slurmctld**
   Es el demonio de control que actúa como el coordinador central de los *cluster* Slurm y este se encuentra instalado en el nodo maestro. Gestiona las decisiones de planificación, mantiene la cola de trabajos y supervisa la asignación de recursos a los trabajos enviados [22]. Dependiendo de las necesidades pueden configurarse uno o varios administradores de respaldo.

2. **Slurmd**
   Este demonio se encuentra instalado en todos los nodos del *cluster* donde se ejecutan los trabajos. Este demonio se encarga de monitorear el uso de recursos del nodo, informar sobre el estado del nodo y aplica límites de recursos para los trabajos en ejecución en ese nodo [22].

3. **Slurmdbd**
   Es un demonio que se emplea para centralizar la información de dos o más *clusters*, administrados por Slurm, en una única base de datos [22]. Este demonio a diferencia de los dos anteriores es opcional.

4. **Herramientas de usuario**
   Estos son todos los comandos que Slurm facilita al usuario para gestionar y monitorear las distintas tareas que se llevan a cabo [22]. Entre estas herramientas se puede destacar:

   - `srun`: Este comando se utiliza para iniciar los trabajos.
   - `scancel`: Este comando se utiliza para cancelar trabajos en ejecución o en espera.
   - `squeue`: Este comando se utiliza visualizar el estado de las tareas.
   - `sacct`: Este comando se utiliza para obtener información más precisa del estado de las tareas en ejecución.
   - `sview`: Este comando se utiliza para cancelar ver de manera visual el estado de un trabajo, además de la topología de red.
   - `scontrol`: Este comando se utiliza para modificar y/o supervisar el estado de un nodo en el *cluster*.
   - `sacctmgr`: Este comando se utiliza para administrar la base de datos.

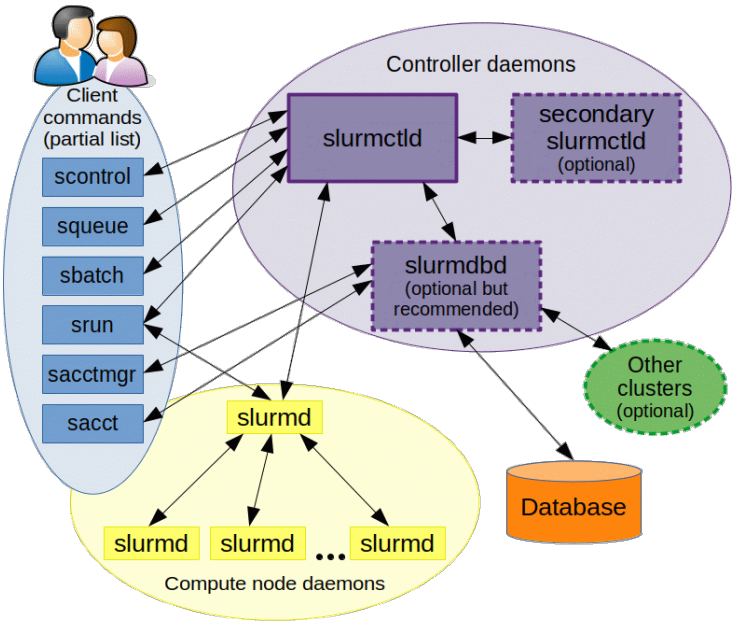


Figura 17: Diagrama con los componentes de Slurm [22].

CAPÍTULO 7

Configuración de *cluster* Ethernet

En la Figura 18 se puede observar un diagrama con la topología del *cluster* Ethernet, que se elaboró. Dicha topología cuenta con un nodo maestro y 2 nodos esclavos, además, de tener una unidad de almacenamiento externa conectada al nodo maestro. Esta unidad de almacenamiento externa se utiliza para almacenar el contenido compartido entre los nodos del *cluster* Ethernet. Este contenido compartido es accesible por medio de la red local ya que, el nodo maestro utiliza el protocolo NFS para exportar la información de la unidad de almacenamiento externa.

El nodo maestro tiene acceso a Internet por medio de su interfaz Wifi (wlan0), y tiene configurado un servidor DHCP en su interfaz Ethernet (eth0), la cual además realiza una traducción de direcciones de red (NAT) entre sus interfaces de red. Todo esto con el fin de que los nodos esclavos puedan acceder a Internet por medio del nodo maestro y con ello facilitar la instalación de las dependencias necesarias. Otro aspecto a destacar es que el nodo maestro configura a los nodos esclavos por medio de un código en lenguaje Python (ver Cuadro 34), el cual se ejecuta cada vez que un nodo esclavo se conecta a la red local y hace una solicitud de DHCP.

Por último, es necesario destacar que el nodo maestro utiliza el protocolo de Slurm para realizar la administración de los recursos del *cluster* Ethernet. La configuración de Slurm y de la red local se detalla en la sección 7.1. En dicha sección se muestran las dependencias necesarias para el correcto funcionamiento de Slurm y de los archivos de configuración utilizados para el *cluster* Ethernet.

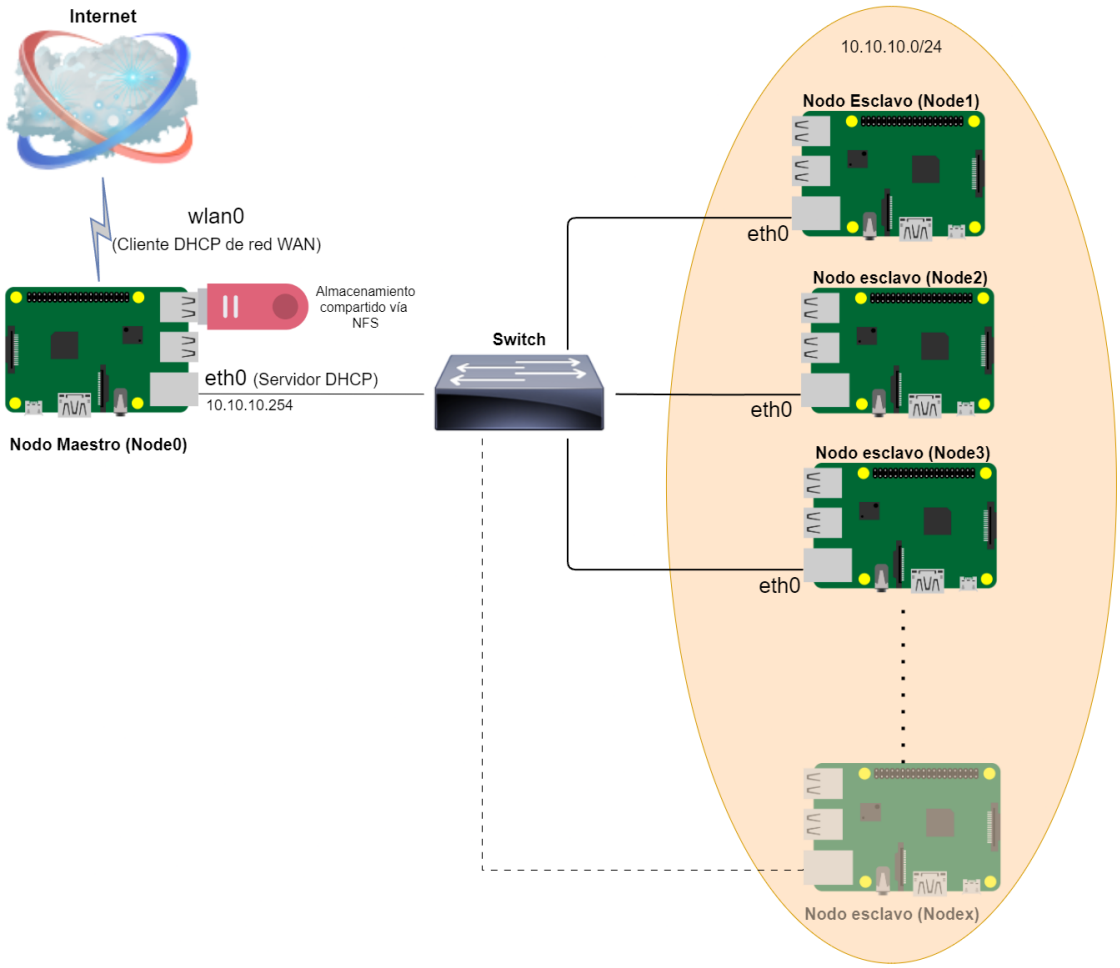


Figura 18: Diagrama de topología de *cluster* Ethernet.

7.1. Configuración de nodo maestro

7.1.1. Configuración de red LAN

1. Actualizar las dependencias de la Raspberry Pi:

```
sudo apt-get update
sudo apt-get upgrade -y
```

Cuadro 2: Comandos para actualizar las dependencias de la Raspberry Pi.

2. Configurar el nombre del nodo maestro:

   a) Editar archivo hostname con permisos de administrador utilizando nano:

```
sudo nano /etc/hostname
```

Cuadro 3: Comando para editar el archivo `/etc/hostname`.

b) Reemplazar el nombre actual por el nombre deseado y guardar los cambios:

```
node0
```

Cuadro 4: Nombre del nodo maestro.

c) Editar archivo hosts con permisos de administrador utilizando nano:

```
sudo nano /etc/hosts
```

Cuadro 5: Comando para editar el archivo `/etc/hosts`.

d) Reemplazar el nombre actual asociado a la dirección 127.0.1.1 por el nombre seleccionado y guardar los cambios:

```
127.0.1.1      node0
```

Cuadro 6: Asociar el nombre del nodo maestro a la dirección IP.

e) Reiniciar la Raspberry Pi:

```
sudo reboot
```

Cuadro 7: Comando para reiniciar la Raspberry Pi.

3. Configurar la dirección IP en la interfaz eth0:

a) Editar archivo dhcpcd.conf con permisos de administrador utilizando nano:

```
sudo nano /etc/dhcpcd.conf
```

Cuadro 8: Comando para editar archivo `/etc/dhcpcd.conf`.

b) Agregar al archivo las siguientes líneas de configuración y guardar los cambios:

```
interface eth0
static ip_address=10.10.10.254/24
static domain_name_servers=8.8.8.8
```

Cuadro 9: Editar archivo `/etc/dhcpcd.conf`.

Para este caso se configuró la última dirección disponible de la red 10.10.10.0/24 y se utilizó el servidor DNS de Google. Cabe resaltar que el servidor DNS puede modificarse por cualquier otro.

c) Para hacer efectivos los cambios en el archivo de dhcpcd.conf es necesario reiniciar el servicio de dhcpcd:

```
sudo systemctl restart dhcpcd
```

Cuadro 10: Comando para reiniciar el servicio de dhcpcd.

d) Para comprobar que la dirección IP fue correctamente configurado:

```
ifconfig
```

Cuadro 11: Comando para comprobar la configuración de la dirección IP.

Se debe resaltar que el puerto Ethernet de la Raspberry Pi debe de encontrarse conectado para que la interfaz se encuentre arriba y pueda apreciarse los cambios.

4. Configurar la conexión entre las interfaces Wifi y Ethernet:

a) Editar archivo sysctl.conf con permisos de administrador utilizando nano:

```
sudo nano /etc/sysctl.conf
```

Cuadro 12: Comando para editar archivo `/etc/sysctl.conf`.

b) Agregar al archivo la siguiente línea de configuración y guardar los cambios:

```
net.ipv4.ip_forward=1
```

Cuadro 13: Configurar persistencia entre la conexión de interfaces.

5. Instalar dnspmasq:

```
sudo apt install dnsmasq -y
```

Cuadro 14: Comando para instalar `dnspmasq`.

6. Configurando el servidor de DHCP

a) Editar archivo dnsmasq.conf con permisos de administrador utilizando nano:

```
sudo nano /etc/dnsmasq.conf
```

Cuadro 15: Comando para editar archivo `/etc/dnsmasq.conf`.

b) Agregar al archivo las siguientes líneas de configuración y guardar los cambios:

```
interface=eth0
dhcp-range=10.10.10.1,10.10.10.200,12h
```

Cuadro 16: Configuración del servidor de DHCP.

Para este caso se restringió la cantidad de direcciones IP que puede otorgar el servidor de DHCP siendo estas desde la 10.10.10.1 hasta la 10.10.10.200, las direcciones en el rango de la 10.10.10.201 hasta la 10.10.10.253 se encuentran disponibles para direccionamiento estático. Cabe resaltar que la restricción de direcciones asignables puede modificarse dependiendo de las necesidades del usuario.

7. Habilitar servicio de dnspmasq:

```
sudo systemctl enable dnsmasq
sudo systemctl start dnsmasq
```

Cuadro 17: Comando para habilitar el servicio de `dnspmasq`.

8. Configuración de NAT entre las interfaces Wifi y Ethernet:

a) Para habilitar el NAT entre las interfaces:

```
sudo iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE
```

Cuadro 18: Comando para habilitar el NAT.

b) Para hacer los cambios persistentes entre cada reinicio:

```
sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"
```

Cuadro 19: Comando para hacer persistente los cambios.

7.1.2. Configuración de Slurm

1. Instalación de los paquetes relacionados a Slurm:

```
sudo apt install -y ntpdate nfs-kernel-server slurm-wlm
```

Cuadro 20: Comando para instalar las dependencias de Slurm.

2. Creación de directorio donde se encontrara el contenido compartido:

```
sudo mkdir /clusterfs
sudo chown nobody.nogroup /clusterfs
sudo chmod -R 777 /clusterfs
```

Cuadro 21: Creación y configuración de usuarios y permisos del directorio compartido.

3. Configurar la unidad de almacenamiento externo:

   a) Identificar el nombre de la unidad de almacenamiento:

```
lsblk
```

Cuadro 22: Comando para identificar todos los dispositivos.

   b) Formatear la unidad de almacenamiento como una partición ext4:

```
sudo mkfs.ext4 /dev/sda1
```

Cuadro 23: Comando para crear particiones `ext4`.

4. Configurar la ubicación de almacenamiento de la carpeta donde se encuentra el contenido compartido:

   a) Encontrar el UUID de la unidad de almacenamiento externo:

```
blkid
```

Cuadro 24: Comando para identificar el UUID de los dispositivos.

   b) Editar archivo fstab con permisos de administrador utilizando nano:

```
sudo nano /etc/fstab
```

Cuadro 25: Editar archivo `/etc/fstab`.

   c) Agregar al archivo la siguiente línea de configuración y guardar los cambios:

```
UUID=<UUID de la unidad externa> /clusterfs ext4 defaults 0 2
```

Cuadro 26: Configurando la ubicación de montaje de la carpeta compartida.

   d) Montar la configuración recién creada:

```
sudo mount -a
```

Cuadro 27: Comando para montar todos los dispositivos descritos en `fstab`.

5. Configurar el directorio compartido, para que se exporte vía NFS:

   a) Editar archivo /etc/exports con permisos de administrador utilizando nano:

```
sudo nano /etc/exports
```

Cuadro 28: Editar archivo `/etc/exports`.

   b) Agregar al archivo la siguiente línea de configuración y guardar los cambios:

```
/clusterfs 10.10.10.0/24(rw,sync,no_root_squash,no_subtree_check)
```

Cuadro 29: Configurando la exportación de la carpeta compartida.

   c) Exportar la configuración recién creada:

```
sudo exportfs -a
```

Cuadro 30: Comando para exportar todos los dispositivos descritos en `exports`.

Una vez instalado todas las dependencias necesarais de Slurm, se procede a editar el archivo de slurm.conf (ver Cuadro 33). Para el *cluster* creado se habilitó el uso del archivo de cgroup, en dicho archivo se configura la utilización de los recursos al momento de realizarse las tareas agendas por Slurm. En el archivo de cgroup.conf (ver Cuadro 31) se definió el uso de un archivo que contiene todos los dispositivos a los cuales los nodos pueden acceder al momento de realizar sus tareas (ver Cuadro 32).

```
# Ruta del punto de montaje de cgroups en el sistema de archivos.
CgroupMountpoint="/sys/fs/cgroup"

# Se habilita el montaje automático de cgroups al iniciar.
CgroupAutomount=yes

# Se define la ruta al directorio que contiene los agentes de liberación para
# cgroups.
CgroupReleaseAgentDir="/etc/slurm/cgroup"

# Se define la ruta al archivo que contiene la lista de dispositivos permitidos
# en cgroups.
AllowedDevicesFile="/etc/slurm/cgroup_allowed_devices_file.conf"

# Se define la restricción de de los núcleos asignados a los trabajos.
ConstrainCores=no

# Se define que no existe aplicación por afinidad de tareas para los trabajos.
TaskAffinity=no

# Se restringe el espacio de memoria RAM para los trabajos.
ConstrainRAMSpace=yes

# No se restringe el espacio de intercambio (swap) para los trabajos.
ConstrainSwapSpace=no

# No se restringen los dispositivos permitidos para los trabajos.
ConstrainDevices=no

# Se define el espacio de memoria RAM permitido para los trabajos, en porcenta-
# je.
AllowedRamSpace=100

# Espacio de intercambio (swap) permitido para los trabajos en porcentaje.
AllowedSwapSpace=0

# Porcentaje máximo de uso de memoria RAM permitido para los trabajos.
MaxRAMPercent=100

# Porcentaje máximo de uso de espacio de intercambio (swap) permitido para los
# trabajos.
MaxSwapPercent=100

# Espacio de memoria RAM mínimo requerido para los trabajos en porcentaje.
MinRAMSpace=30
```

Cuadro 31: Archivo de configuración de Cgroup.

```
# Dispositivo nulo, utilizado para descartar datos escritos en él.
/dev/null

# Generador de números aleatorios no bloqueante.
/dev/urandom

# Generador de secuencias de ceros.
/dev/zero

# Dispositivos correspondientes al disco duro sda.
/dev/sda*

# Dispositivos de CPU, que pueden incluir núcleos individuales y threads.
/dev/cpu/*/*

# Dispositivos de pseudo terminales, utilizados para la comunicación con termi-
# nales virtuales.
/dev/pts/*

# Dispositivos correspondientes a recursos de clusterfs.
/clusterfs*
```

Cuadro 32: Archivo de configuración de dispositivos permitidos para Cgroup.

```
# *Configuración del controlador de Slurm*
# Nombre del host donde se ejecutará el controlador Slurm.
SlurmctldHost=node0
# Puerto en el que el controlador Slurm escuchará para conexiones entrantes.
SlurmctldPort=6817
# Ruta al archivo que almacena el PID del controlador.
SlurmctldPidFile=/run/slurmctld.pid
# Tiempo de espera, en segundos, para la comunicación con el controlador.
SlurmctldTimeout=120
# Nivel de depuración para el controlador (info muestra información detallada).
SlurmctldDebug=info
# Ruta al archivo de registro del controlador.
SlurmctldLogFile=/var/log/slurm/slurmctld.log

# *Usuario para ejecutar procesos de Slurm*
# Usuario bajo el cual se ejecutarán los procesos de Slurm.
SlurmUser=slurm

# *Ubicación para guardar información de estado y trabajos*
StateSaveLocation=/var/lib/slurm/slurmctld

# *Tipo de seguimiento de procesos*
# Se configura el uso de cgroups.
ProctrackType=proctrack/cgroup

# *Almacenamiento y manejo de la contabilidad (accounting) de trabajos*
# Se desactiva el almacenamiento de contabilidad.
AccountingStorageType=accounting_storage/none
# Almacenar comentarios de trabajos en registros de contabilidad.
AccountingStoreJobComment=YES

# *Nombre del cluster Slurm*
# Se configura el nombre de cluster como PiCluster
ClusterName=PiCluster

# *Tipo de programador (scheduler)*
# Se utiliza el programador de retroceso (backfill).
SchedulerType=sched/backfill

# *Tipo de selección de recursos y sus parámetros*
# Se configura las restricciones de recursos.
```

```
SelectType=select/cons_res
# Se configura los parámetros específicos para la selección de recursos (Core
# Resources).
SelectTypeParameters=CR_Core

# *Tipo de conmutador (switch)*
# Se configura como ninguno (none)
SwitchType=switch/none

# *Plugin para la afinidad de tareas*
# Se configura el plugin por afinidad.
TaskPlugin=task/affinity

# *Configuración de los temporizadores*
# Tiempo en segundos antes de que un trabajo inactivo se elimine.
InactiveLimit=0
# Tiempo en segundos antes de matar un trabajo después de enviar una señal de
# terminación.
KillWait=30
# Tiempo en segundos antes de que un trabajo sea elegible para la terminación
# debido a su edad.
MinJobAge=300
# Tiempo en segundos que se debe esperar antes de volver a lanzar trabajos can-
# celados.
Waittime=0

# *Configuración de recopilación de información de trabajos*
# Frecuencia en segundos para recopilar información de trabajos.
JobAcctGatherFrequency=30
# Tipo de recopilación de información de trabajos.
JobAcctGatherType=jobacct_gather/none

# *Otras configuraciones*
MpiDefault=none  # Opción predeterminada para MPI (Message Passing Interface).
# Indica si los nodos vuelven al servicio después de una des-
# conexión.
ReturnToService=1
JobCompType=jobcomp/none  # Tipo de recopilador de información de trabajos.

# *Configuración de los nodos de cómputo*
# NodeName: Nombre del nodo de cómputo.
# NodeAddr: Dirección IP del nodo de cómputo.
# CPUs: Número de CPUs en el nodo.
# State: Estado actual del nodo de cómputo.
NodeName=node0 NodeAddr=10.10.10.254 CPUs=4 State=UNKNOWN

# *Configuración de las particiones (partitions)*
# PartitionName: Nombre de la partición.
# Nodes: Nodos incluidos en esta partición.
# Default: Indica si esta partición es la partición predeterminada (YES o NO).
# MaxTime: Tiempo máximo de ejecución permitido para los trabajos en esta
# partición.
# State: Estado de la partición (UP, DOWN, DRAIN, ...).
PartitionName=world Nodes=node[0] Default=YES MaxTime=INFINITE State=UP
```

Cuadro 33: Archivo de configuración de Slurm.

7.2. Código para configuración de nodos esclavos

Para agilizar el proceso de configuración de los nodos esclavos, se decidió crear un código en lenguaje Python (ver Cuadro 34) el cual se encarga de instalar todas las dependencias

y paquetes necesarios para un correcto funcionamiento, además, se asegura de configurar tanto el nombre del nodo como los parámetros de red y también de la configuración de Slurm para permitir que el nodo esclavo pueda ser parte del *cluster* Ethernet. El código se ejecuta cada vez que una Raspberry Pi 3B se conecte al servidor de DHCP y esta hace una solicitud hacia este, además, el código se encarga de llevar un registro de los nodos esclavos que se conectan al servidor de DHCP y ya se encuentran configurados, para evitar que el nodo vuelva a pasar por el proceso de configuración.

```
################################################################################
# Autor: Daniel Mundo                                                          #
# Fecha: 2023-01-06                                                            #
# Descripción: Este script se encarga de configurar un nuevo nodo al cluster. #
################################################################################
#                                 Librerías                                    #
################################################################################
import paramiko          # Librería para el manejo de la conexión SSH.
import sys               # Librería para obtener los parámetros de entrada.
import re                # Librería para el uso de expresiones regulares.
import time              # Librería para realizar delays dentro del código.
import subprocess        # Librería para ejecutar comandos en el sistema local.
import logging           # Librería para almacenar información del código.
import json              # Librería para el manejo de archivos JSON.
################################################################################
#                                 Constantes                                   #
################################################################################
# Nombre de usuario para la conexión SSH.
USER_NAME = 'pi'
# Contraseña que se utiliza para la conexión SSH.
PASSWORD  = 'raspberrypi'
# Directorio general del cluster.
DIRECTORIO_GENERAL = '/clusterfs'
# Directorio donde se almacenan todos los archivos de configuración de SLurm.
DIRECTORIO_CONFIG = DIRECTORIO_GENERAL + '/docs/config'
# Directorio donde se almacenan los archivos de Slurm en el nodo.
DIRECTORIO_SLURM = '/etc/slurm'
# Directorio donde se almacenan los archivos log de configuración de nodos.
DIRECTORIO_LOG = DIRECTORIO_GENERAL + '/docs/logs/new_node'
# Archivo que almacena la información de los hosts.
ARCHIVO_HOSTS = '/etc/hosts'
# Archivo que auxiliar que almacena la información de los hosts.
ARCHIVO_HOSTS_AUX = DIRECTORIO_CONFIG + '/hosts_aux'
# Archivo donde se almacena la llave de munge.
ARCHIVO_MUNGE_KEY = '/etc/munge/munge.key'
# Archivo de configuración de Slurm (Backup), ubicado en el directorio común.
ARCHIVO_SLURM_CONF_BACKUP = DIRECTORIO_CONFIG + '/Backup_slurm.conf'
# Archivo de paquetes a instalar en el nuevo nodo.
ARCHIVO_PAQUETES = DIRECTORIO_CONFIG + '/paquetes.txt'
################################################################################
#                                Funciones                                     #
################################################################################
def leer_estado_nodo(ruta_json='', nombre_nodo='', **info_nodo):
    """ Esta función se encarga de comprobar si el nodo se encuentra guardado
    en el archivo JSON y el estado del nodo.
    Argumentos:
        ruta_json {str} -- Es la ruta donde se encuentra almacenado el archivo
        JSON. (Por defecto: {''})
        nombre_nodo {str} -- Es el nombre que se la asigna al nodo.
        (Por defecto: {''})
        **info_nodo {dict} -- Es un diccionario que contiene la información del
        nodo que se desea almacenar. (Por defecto: {{}})
    Retorna:
        banderas {list[bool]} -- Es una lista que contiene las banderas de
        estado del nodo. La primera bandera indica si el nodo se encuentra
        guardado en el archivo JSON. La segunda bandera indica si el nodo se
        encuentra disponible.
    """
```

```
    # Se inicializan las banderas de estado.
    guardado = False    # Indica si el nodo se encuentra guardado en el archivo.
    disponible = False  # Indica si el nodo se encuentra disponible.

    # Se lee el archivo JSON
    with open(ruta_json, 'r') as archivo:
        info= json.load(archivo) # Se guarda en la variable info.

    # Se revisa si el nodo se encuentra guardado en el archivo JSON.
    if nombre_nodo not in info:
        # Se procede ha almacenar la información del nodo en la variable info.
        info[nombre_nodo] = info_nodo
        # Se procede a modificar el archivo JSON.
        with open(ruta_json, 'w') as archivo:
            json.dump(info, archivo, indent=4)
        return [guardado, disponible]       # Se retornan las banderas.
    else:
        guardado = True
        # Se revisa si el nodo se encuentra disponible.
        if info[nombre_nodo]['status'] == 1:
            disponible = True               # Se cambia el estado de la bandera.
            return [guardado, disponible]   # Se retornan las banderas.
        else:
            # Se procede a modificar el estado del nodo.
            info[nombre_nodo]['status'] = 1 # Se cambia el estado del nodo.
            # Se procede a modificar el archivo JSON.
            with open(ruta_json, 'w') as archivo:
                json.dump(info, archivo, indent=4)
            return [guardado, disponible]   # Se retornan las banderas.
def generar_comandos_de_instalacion(ruta_archivo=''):
    """Genera una lista de comandos de instalación de paquetes Linux.
    Argumentos:
        ruta_archivo {str} -- Ruta del archivo de texto con los paquetes a
        instalar. (Por defecto: {''})
    Retorna:
        lista_de_comandos {list} -- Lista de comandos de instalación de
        paquetes Linux.
    """
    # Se lee el archivo de texto y se almacena en una lista.
    with open(ruta_archivo, 'r') as archivo:
        lineas = archivo.readlines()
    # Se divide el archivo en secciones, cada sección es una lista.
    secciones = ''.join(lineas).split('#')[1:]
    secciones = [seccion.split('\n')[1:-1] for seccion in secciones]
    lista_de_comandos = [] # Lista de comandos de instalación.
    # Para cada sección, generar un comando de instalación de paquetes Linux.
    for seccion in secciones:
        paquetes = ' '.join(seccion) # Unir los paquetes de la sección.
        # Generar el comando de instalación y agregarlo a la lista de comandos.
        lista_de_comandos.append(f'apt-get install -y {paquetes}')
    return lista_de_comandos # Devolver la lista de comandos de instalación.
def list_to_str(lista_de_elementos=[], sep='\n'):
    """ Esta función se encarga de convertir una lista de elementos en una
    cadena de caracteres.
    Argumentos:
        lista_de_elementos {list} --  Almacena los elementos que se desean
        convertir en una cadena de caracteres. (Por defecto: {[]})
        sep {str} -- Es el separador que se emplea para dividir los elementos
        en la cadena de caracteres final. (Por defecto: {'\n'})
    Retorna:
        cadena_a_retornar {str} -- Es la cadena de caracteres que contiene los
        elementos de la lista divididos por el separador especificado.
    """
    cadena_a_retornar = ''  # Variable que almacena el string final
    # Itera a través de los elementos de la lista y los concatena al string
    # final utilizando el separador especificado.
    for elemento in lista_de_elementos:
        cadena_a_retornar += str(elemento) + sep
```

```
    return cadena_a_retornar  # Retorna la cadena de caracteres final.
def actualizar_archivos_Slurm(node_name='', node_ip=''):
    """Esta función se encarga de actualizar los archivos de configuración de
    Slurm.
    Argumentos:
        node_name {str} -- Es el nombre del nodo. (default: {''})
        node_ip {str} -- Es la dirección IP del nodo. (default: {''})
    Returns:
        {list} -- Retorna una lista con los siguientes elementos:
            [0] {str} -- Contiene la información actualizada del archivo
            slurm.conf.
            [1] {str} -- Contiene la información original del archivo
            slurm.conf para backup.
            [2] {list} -- Contiene la información actualizada del archivo
            hosts.
            [3] {str} -- Contiene el ID de los nodos.
    """
    with open(DIRECTORIO_CONFIG+'/slurm.conf', 'r') as file:
        # Obteniendo la información actual del archivo slurm.conf
        file_slurm_conf = file.read()
    # Contiene información incluida en los archivos de hosts.
    list_ip_hostname = []
    # Contiene las filas con información de los nodos
    list_node_info = []
    list_node_info_split = [] # Contiene los elementos de cada fila separados.
    list_index_row = [] # Contiene el índice de las filas con info de los nodos.
    list_node_id = []   # En esta lista se almacena el ID de los nodos.
    # Lista con las filas del archivo slurm.conf
    list_slurm_conf = file_slurm_conf.split('\n')
    # Itera a través de las filas del archivo slurm.conf
    for i, row in enumerate(list_slurm_conf):
        if row:
            if (row[0] == 'N'):
                # Obteniendo las filas con información de los nodos y separando
                # los elementos.
                list_node_info_split.append(re.split(r'[= ]', row))
                # Almacenando las filas con información de los nodos.
                list_node_info.append(row)
                # Almacenando el índice de las filas con información de los
                # nodos.
                list_index_row.append(i)
    # Almacena la relación entre los hostnames y sus respectivas IPs.
    list_ip_hostname = ["{}\t\t{}\n".format(row[3], row[1])
                        for row in list_node_info_split]
    # Almacena el ID de todos los hostnames previamente configurados.
    list_node_id = [int(re.split(r'(\d+)', row[1])[-2])
                    for row in list_node_info_split]
    # Agregando la información del nodo actual
    node_info = "NodeName={} NodeAddr={}".format(node_name, node_ip)
    node_info += " CPUs=4 State=UNKNOWN"
    node_id = node_ip.split('.')[-1]
    list_node_info.append(node_info)  # Agregar al archivo slurm.conf
    list_node_id.append(int(node_id))  # Última línea del archivo slurm.conf
    # Se agrega la relación de hostname y IP del nuevo nodo
    list_ip_hostname.append("{}\t\t{}".format(node_ip, node_name))
    # Crear un string que contiene los IDs separados por comas
    str_node_id = '[{}]'.format(list_to_str(list_node_id, ',')[:-1])
    # Crear la última línea del archivo slurm.conf
    lst_line_slurm_conf = f"PartitionName=world Nodes=node{str_node_id}"
    lst_line_slurm_conf += " Default=YES MaxTime=INFINITE State=UP"
    # Contiene la información original de slurm.conf para backup
    backup_text_slurm_conf = file_slurm_conf
    # Editar la información que se colocará en slurm.conf
    new_list_slurm_conf = list_slurm_conf[:list_index_row[0]]
    new_list_slurm_conf += list_node_info + [lst_line_slurm_conf]
    # Convertir la lista con la información a un string
    new_text_slurm_conf = list_to_str(new_list_slurm_conf)
    # Retornar la información de los archivos slurm.conf y hosts
```

```
195     return [new_text_slurm_conf, backup_text_slurm_conf,
196             list_ip_hostname, str_node_id]
197 def configurar_archivos_log(nuevo_nodo=''):
198     """ Esta función se encarga de configurar los archivos de logs.
199     Argumentos:
200         nuevo_nodo {str} -- Contiene el nombre del archivo de logs.
201         (default: {''})
202     Retorna:
203         {list[logger]} -- Es una lista que contiene los objetos de logs, el
204         primer objeto contiene los logs del nuevo nodo y el segundo objeto
205         contiene los logs del nodo maestro.
206     """
207     # Se define el formato de los logs.
208     formato_log = '%(asctime)s|%(name)s (%(levelname)s): %(message)s'
209     # Se crea el objeto de logs con el nombre del nuevo nodo.
210     logger_nn = logging.getLogger(nuevo_nodo)
211     # Se configura el nivel de logs.
212     logger_nn.setLevel(logging.INFO)
213     # Se crea el objeto de logs con el nombre del nodo maestro.
214     logger_mn = logging.getLogger('node0')
215     # Se configura el nivel de logs.
216     logger_mn.setLevel(logging.INFO)
217     # Se configura el formato de los logs.
218     formato = logging.Formatter(formato_log)
219     #* Se configura el manipulador de logs. *#
220     # Se configura el primer manipulador de logs.
221     manipulador = logging.FileHandler(DIRECTORIO_LOG + f'/{nuevo_nodo}.log')
222     # Se configura el nivel del primer manipulador de logs.
223     manipulador.setLevel(logging.INFO)
224     # Se agrega el formato al manipulador de logs.
225     manipulador.setFormatter(formato)
226     # Se agrega el manipulador de logs al objeto de logs del nuevo nodo.
227     logger_nn.addHandler(manipulador)
228     # Se agrega el manipulador de logs al objeto de logs del maestro.
229     logger_mn.addHandler(manipulador)
230     return [logger_nn, logger_mn] # Se retornan los objetos de logs.
231 def ejecutar_comandos_remotos(cliente_ssh = paramiko.SSHClient(),
232                               log=logging.getLogger(), comando=''):
233     """Esta función se encarga de ejecutar comandos en un cliente SSH, además
234     de verificar si el comando requiere de la contraseña para acceder a los
235     permisos de super usuario.
236     Argumentos:
237         cliente_ssh {SSHClient} -- Contiene la conexión SSH con el nodo.
238         (default: {paramiko.SSHClient()})
239         log {logger} -- Contiene el objeto de logs.
240         (default: {logging.getLogger()})
241         comando {str} -- Comando que se desea ejecutar en el cliente SSH.
242         (default: {''})
243     """
244     # Se comprueba si el comando requiere de la contraseña para acceder a los
245     # permisos de super usuario.
246     _, stdout, stderr = cliente_ssh.exec_command('sudo -n true')
247     # Se almacena la salida de la ejecución del comando.
248     error_output = stderr.read().decode()
249     # Se comprueba si el comando requiere de la contraseña para acceder a los
250     # permisos de super usuario.
251     if 'a password is required' in error_output:
252         # Si el comando requiere de la contraseña, se agrega al inicio del
253         # comando y se utiliza la opción -S para que el comando pueda leer la
254         # contraseña desde la entrada estándar.
255         comando = f'echo {PASSWORD} | sudo -S {comando}'
256     else:
257         # Si el comando no requiere de la contraseña, se agrega el prefijo
258         # sudo al inicio del comando.
259         comando = f'sudo {comando}'
260     # Se toma el tiempo de inicio de la ejecución del comando.
261     tiempo_inicial = time.time()
262     log.info(f"Se ejecutará el comando: {comando}")
```

```
    # Se ejecuta el comando en el cliente SSH.
    _, stdout, stderr = cliente_ssh.exec_command(comando)
    # Se obtiene el tiempo de finalización de la ejecución del comando.
    tiempo_de_ejecucion = time.time() - tiempo_inicial
    # Se muestra la salida del comando.
    if stdout.read().decode() != '':
        log.info(f"Salida del comando:{stdout.read().decode()}")
    if stderr.read().decode() != '':
        # Se muestra la salida de error del comando.
        log.error(f"Error del comando:{stderr.read().decode()}")
        # Se muestra el mensaje de finalización de la ejecución del comando.
        log.info(f"El comando fallo a los {tiempo_de_ejecucion} segundos.")
    else:
        # Se muestra el mensaje de finalización de la ejecución del comando.
        log.info(f"Se ejecuto correctamente en {tiempo_de_ejecucion} segundos.")
    return
def ejecutar_comando_bash(comando='', log=logging.getLogger()):
    """Esta función se encarga de ejecutar un comando bash en el sistema local.
    Argumento:
        command {str} -- Comando bash que se desea ejecutar.
        log {logger} -- Contiene el objeto de logs.
    """
    # Se toma el tiempo de inicio de la ejecución del comando.
    tiempo_inicial = time.time()
    # Se muestra el comando que se va a ejecutar.
    log.info(f"Se ejecutará el comando:\n{comando}")
    try:
        # Ejecuta el comando en una nueva subshell (shell secundaria).
        # -'shell=True' indica que se debe usar la shell del sistema operativo.
        # -'check=True': hace que se genere una excepción si el comando
        # devuelve un código de retorno no cero.
        # -'stdout=subprocess.PIPE': redirige la salida estándar del proceso
        # a una tubería.
        # -'stderr=subprocess.PIPE': redirige la salida de error estándar del
        # proceso a una tubería.
        # -'text=True': hace que los flujos de salida sean tratados como texto.
        result = subprocess.run(comando, shell=True,
                                check=True, stdout=subprocess.PIPE,
                                stderr=subprocess.PIPE, text=True)
        # Se obtiene el tiempo de ejecución del comando.
        tiempo_de_ejecucion = time.time() - tiempo_inicial
        # Se imprime la salida estándar del comando.
        msg_info = f"Tiempos de ejecución: {tiempo_de_ejecucion} segundos."
        msg_info += f"\nSalida estándar:\n{result.stdout}"
        msg_info += f"\nSalida de error:\n{result.stderr}"
        log.info(msg_info)
        return
    except subprocess.CalledProcessError as e:
        # Se obtiene el tiempo de ejecución del comando.
        tiempo_de_ejecucion = time.time() - tiempo_inicial
        # Se imprime la salida estándar del comando.
        msg_error = f"Tiempos de ejecución: {tiempo_de_ejecucion} segundos."
        msg_error += f"\nSalida estándar:\n{e.stdout}"
        msg_error += f"\nSalida de error:\n{e.stderr}"
        log.error(msg_error)
        return
def esperar_reinicio(hostname, username, password, timeout=180):
    """Esta función se encarga de esperar a que el nodo reinicie y se pueda
    establecer una conexión SSH.
    Argumentos:
        hostname {_type_} -- Es la dirección IP del nodo.
        username {_type_} -- Es el nombre de usuario para la conexión SSH.
        password {_type_} -- Es la contraseña para la conexión SSH.
        timeout {int} -- Es el tiempo máximo que se espera para establecer una
        conexión SSH. (default: {180})
    Returns:
        {bool} -- Retorna True si se establece una conexión SSH, caso contrario
        retorna False.
```

```
331     """
332     start_time = time.time()  # Captura el tiempo actual en segundos.
333     # Bucle mientras no se haya superado el tiempo limite.
334     while time.time() - start_time < timeout:
335         try:
336             ssh = paramiko.SSHClient()  # Crea una instancia de cliente SSH.
337             # Configura la política para agregar automáticamente claves de host
338             # desconocidas.
339             ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
340             # Intenta conectarse al servidor SSH.
341             ssh.connect(hostname, username=username, password=password)
342             ssh.close()  # Cierra la conexión SSH.
343             # Retorna True si la conexión se estableció exitosamente.
344             return True
345         # Captura excepción en caso de fallo de autenticación.
346         except paramiko.AuthenticationException:
347             return False  # Retorna False si la autenticación falla.
348         except Exception:  # Captura otras excepciones.
349             time.sleep(5)  # Espera 5 segundos antes de reintentar la conexión.
350     # Retorna False si se agota el tiempo límite sin establecer una conexión
351     # exitosa.
352     return False
353 def main(client_hostname='', client_ip_addr=''):
354     """Esta es la función principal del script, se encarga de configurar el
355     nuevo nodo.
356     Argumentos:
357         client_hostname {str} -- Es el nombre del nuevo nodo. (default: {''})
358         client_ip_addr {str} -- Es la dirección IP del nuevo nodo.
359         (default: {''})
360     """
361     # Se obtiene el tiempo de inicio de la configuración del nuevo nodo.
362     tiempo_inicial = time.time()
363     # Contiene la línea de texto para configurar el directorio común con nfs.
364     mount_common_directory = ('10.10.10.254:/clusterfs\t\t/clusterfs\t\tnfs'
365                               '\t\tdefaults\t\t0 0')
366     # Se coloca como parámetros de la función el hostname, la IP del nodo y la
367     # ruta para obtener slurm.conf del directorio común, como respuesta se
368     [update_slurm_conf, backup_slurm_conf,
369     list_update_hosts, nodes_id] = actualizar_archivos_Slurm(client_hostname,
370                                                              client_ip_addr)
371     # Se eliminan los corchetes y se convierte a una lista de strings.
372     nodes_id = nodes_id[1:-1].split(',')
373     nodes_id_configured = [int(node_id) for node_id in nodes_id]
374     # Se obtiene la cantidad total de nodos ya configurados
375     number_nodes_configured = (len(list_update_hosts)-1)
376     # Se procede recuperar la información contenida en el archivo de hosts.
377     with open(ARCHIVO_HOSTS, 'r') as file:
378         local_hosts_file = file.read()
379     # Se genera una lista la cual contiene cada fila del archivo hosts.
380     list_local_hosts_file = local_hosts_file.split('\n')[:6]
381     # Se edita la línea de que hace referencia al hosts local.
382     list_local_hosts_file[-1] = "127.0.1.1\t\t{}".format(client_hostname)
383     # Se procede agregar todos los host ya configurados (se incluye el master).
384     join_list=list_local_hosts_file+list_update_hosts[:-1]
385     # Se convierte la lista a un string.
386     str_to_load = list_to_str(join_list)
387     del join_list # Se libera la variable join_list.
388     # Se procede a actualizar el archivo auxiliar de hosts.
389     with open(ARCHIVO_HOSTS_AUX, 'w') as file:
390         file.write(str_to_load)
391     # Realiza las actualizaciones e instalaciones necesarias en el nuevo nodo.
392     comandos_instalacion_nuevo_nodo =[
393         'apt-get update',
394         'apt-get upgrade -y'
395         ]
396     comandos_instalacion_nuevo_nodo.extend(
397         generar_comandos_de_instalacion(ARCHIVO_PAQUETES)
398         )
```

```
    # Realiza las configuraciones básicas para poder integrar el nuevo nodo al
    # cluster.
    comandos_configuraciones_basicas= [
        # Configura el hostname del nuevo nodo.
        'hostnamectl set-hostname {}'.format(client_hostname),
        # Crea el directorio común.
        f'mkdir {DIRECTORIO_GENERAL}',
        # Coloca grupo y usuarios que pueden acceder
        f'chown nobody.nogroup -R {DIRECTORIO_GENERAL}',
        # Coloca los permisos
        f'chmod 777 -R {DIRECTORIO_GENERAL}',
        # Agrega fuente del directorio común
        f'echo "{mount_common_directory}" | sudo tee -a /etc/fstab'
    ]
    # Realiza las configuraciones necesarias para el correcto funcionamiento
    # de slurm en el nuevo nodo.
    comandos_configurar_slurm = [
        # Se copia el archivo hosts_aux al archivo hosts del nuevo nodo.
        f'cp {ARCHIVO_HOSTS_AUX} {ARCHIVO_HOSTS}',
        # Copia el archivo slurm.conf del directorio común a /etc/slurm/
        f'cp {DIRECTORIO_CONFIG}/slurm.conf {DIRECTORIO_SLURM}/slurm.conf',
        # Copia el archivo munge.key del directorio común a /etc/munge/
        f'cp {DIRECTORIO_CONFIG}/munge.key {ARCHIVO_MUNGE_KEY}',
        # Copia todo los archivos cgroup del directorio común a /etc/slurm/
        f'cp {DIRECTORIO_CONFIG}/cgroup* {DIRECTORIO_SLURM}',
        # Habilita el demonio de munge
        'sudo systemctl enable munge',
        # Inicia el demonio de munge
        'sudo systemctl start munge',
        # Habilita el demonio de slurm
        'sudo systemctl enable slurmd',
        # Inicia el demonio de slurm
        'sudo systemctl start slurmd'
    ]
    # Realiza las actualizaciones necesarias en los nodos ya configurados para
    # que el nuevo nodo pueda ser utilizado.
    commandos_actualizar_nodos = [
        # Actualiza archivo de host en todos los nodos ya configurados
        'echo "{}" | sudo tee -a {}'.format(list_update_hosts[-1],
                                            ARCHIVO_HOSTS),
        # Actualiza slurm.conf en todos los nodos ya configurados
        'sudo cp -u {} {}'.format(DIRECTORIO_CONFIG + '/slurm.conf',
                                  DIRECTORIO_SLURM  + '/slurm.conf')
    ]
    # Realiza las configuraciones finales para hacer efectivas las
    # configuraciones realizadas en el nuevo nodo.
    comandos_reinicio_daemons= [
        # Reinicia el demonio gestión de slurm
        'sudo systemctl restart slurmctld',
        'sinfo'
    ]
    # Configurando instancia de logging, para almacenar en un archivo log toda
    # la información acerca del nuevo nodo configurado.
    [log_esclavo, log_maestro] = configurar_archivos_log(client_hostname)
    #* Inicio de la primera conexión SSH (con el nuevo nodo)
    ssh = paramiko.SSHClient() # Crea una instancia de cliente SSH
    # Configura la política para agregar automáticamente claves de host
    # desconocidas.
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    # Se conecta al servidor SSH.
    ssh.connect(hostname=client_ip_addr, username=USER_NAME, password=PASSWORD)
    # Se agregan las primeras lineas del archivo que contendrá la información
    # correspondiente a la configuración del nuevo nodo.
    log_maestro.info(f"Se inicia la configuración del nodo {client_hostname}.")
    # Se procede a realizar las primeras configuraciones del nodo las cuales
    # Se realizan las actualizaciones e instalaciones.
    for comando_bash in comandos_instalacion_nuevo_nodo:
        # Se ejecuta el comando en el host remoto.
```

```
467         ejecutar_comandos_remotos(ssh, log_esclavo, comando_bash)
468     # Se realizan las configuraciones básicas.
469     for comando_bash in comandos_configuraciones_basicas:
470         # Se ejecuta el comando en el host remoto.
471         ejecutar_comandos_remotos(ssh, log_esclavo, comando_bash)
472     # Se reinicia el host remoto para hacer efectivas las configuraciones.
473     ejecutar_comandos_remotos(ssh, log_esclavo, 'reboot')
474     with open(DIRECTORIO_CONFIG+'/slurm.conf', 'w') as file:
475         # Se actualiza el archivo de slurm.conf
476         file.writelines(update_slurm_conf)
477     with open(ARCHIVO_SLURM_CONF_BACKUP, 'w') as file:
478         # Se actualiza el backup del archivo de slurm.conf
479         file.writelines(backup_slurm_conf)
480     # Esta condicional se asegura que exista más de un nodo ya configurado para
481     # no generar problemas con los comandos de slurm.
482     if(number_nodes_configured>=1):
483         for id in nodes_id_configured[:-1]:
484             # Se procede a realizar las actualizaciones de configuraciones para
485             # todos los nodos que se encontraban configurados previamente.
486             for bash_command in commandos_actualizar_nodos:
487                 comando_slurm  = f"srun --nodelist=node{id} /bin/bash -c"
488                 comando_slurm += f" '{bash_command} ; exit'"
489                 ejecutar_comando_bash(comando_slurm, log_maestro)
490     # Se procede a realizar intentos de conexión ssh con el nodo reiniciado, en
491     # caso el nodo responda validate_reboot es True, caso contrario False
492     validate_reboot = esperar_reinicio(client_hostname, USER_NAME,  PASSWORD)
493     # Si el host responde, se procede a realizar las configuraciones finales al
494     # nuevo nodo, sino se agrega una línea que notifica la no respuesta del nodo
495     # al reinicio.
496     if(validate_reboot==True):
497         ssh = paramiko.SSHClient() # Crea una instancia de cliente SSH
498         # Configura la política para agregar automáticamente claves de host
499         # desconocidas.
500         ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
501         # Se conecta al servidor SSH.
502         ssh.connect(hostname=client_ip_addr,username=USER_NAME,
503                     password=PASSWORD)
504         # Se procede a realizar las últimas configuraciones del nuevo nodo estas
505         for comando_bash in comandos_configurar_slurm:
506             # Se ejecuta el comando en el host remoto.
507             ejecutar_comandos_remotos(ssh, log_esclavo, comando_bash)
508 
509         # Se procede a esperar 30 segundos antes de reiniciar los demonios
510         # slurm y slurmctl.
511         time.sleep(30)
512         # Se procede a realizar las configuraciones finales del nodo maestro.
513         for comando_bash in comandos_reinicio_daemons:
514             # Se ejecuta el comando en el host remoto.
515             ejecutar_comando_bash(comando_bash, log_maestro)
516         # Se obtiene el tiempo de finalización de la configuración del nuevo
517         # nodo.
518         tiempo_final = time.time() - tiempo_inicial
519         # Se imprime el tiempo de configuración del nuevo nodo.
520         msg_info  = f"Tiempo de configuración del nodo {client_hostname}: "
521         msg_info += f"{tiempo_final} segundos."
522         log_maestro.info(msg_info)
523     else:
524         # Se obtiene el tiempo de finalización de la configuración del nuevo
525         # nodo.
526         tiempo_final = time.time() - tiempo_inicial
527         # Se imprime el tiempo de configuración del nuevo nodo.
528         msg_error  = f"El nodo {client_hostname} no respondió tras reinicio."
529         msg_error += f"Tiempo de configuración del nodo {client_hostname}: "
530         msg_error += f"{tiempo_final} segundos."
531         log_maestro.error(msg_error)
532     return
533 ################################################################################
534 if __name__ == "__main__":
```

```
    ruta_del_archivo_json = '/clusterfs/docs/config/cluster_nodos.json'
    # Se recuperan los argumentos de entrada.
    direccion_IP = sys.argv[1]  # Se recupera la dirección IP del nuevo nodo.
    direccion_MAC = sys.argv[2] # Se recupera la dirección MAC del nuevo nodo.
    # Se genera el nombre del nodo.
    nombre_nodo = 'node' + direccion_IP.split('.')[-1]
    # Se genera el diccionario que contiene la información del nodo.
    info_nodo = {'ip': direccion_IP, 'mac': direccion_MAC, 'status': 0}
    # Se procede a comprobar el estado del nodo.
    [guardado, estado] = leer_estado_nodo(ruta_del_archivo_json,
                                          nombre_nodo, **info_nodo)
    # Se comprueba si el nodo se encuentra guardado en el archivo JSON.
    if guardado:
        # Se comprueba si el nodo se encuentra disponible.
        if estado:
            # Se procede a configurar el nuevo nodo.
            print(f'El nodo {nombre_nodo} se encuentra configurado.')
        else:
            # Se imprime un mensaje.
            print(f'El nodo {nombre_nodo} se esta configurando...')
    else:
        # Se procede a iniciar la configuración del nuevo nodo.
        main(nombre_nodo, direccion_IP)
        # Se actualiza el estado del nodo.
        info_nodo['status'] = 1
        # Se procede a modificar el archivo JSON.
        with open(ruta_del_archivo_json, 'r') as archivo:
            info = json.load(archivo) # Se guarda en la variable info.
        # Se procede a modificar el archivo JSON.
        with open(ruta_del_archivo_json, 'w') as archivo:
            json.dump(info, archivo, indent=4)
```

Cuadro 34: Código de configuración de nodos esclavos.

7.3. Código de multiplicación de matrices

El código en lenguaje Python (ver Cuadro 35) se implementó para llevar a cabo las pruebas de rendimiento, las cuales implicaban el realizar multiplicaciones de matrices cuadras y con ello obtener muestras para comparar los tiempos de ejecución entre los diferentes protocolos de comunicación.

Al momento de ejecutarse el código, genera dos matrices de tamaño $n \times n$, donde n es un número entero dado por el usuario como parámetro al momento de ejecutar el código. Una vez generadas las matrices, el código se encarga de entregar la matriz B y una parte de la matriz A a cada uno de los diferentes procesos empleando las funciones de la librería de mpi4py. Posteriormente, el código realiza la multiplicación de matrices, para finalmente, imprimir la matriz resultante y el tiempo total de ejecución de cada uno de los procesos involucrados en la multiplicación de matrices.

```
#############################################################################
# Autor: Daniel Mundo                                                       #
# Fecha: 2023-01-06                                                         #
# Descripción: Código para realizar la multiplicación de matrices utilizando  #
# MPI. El código recibe como argumentos las dimensiones de las matrices y     #
# como parámetros opcionales el número de prueba, la cantidad de nodos y la   #
# cantidad de procesos por nodo.                                            #
# Uso:                                                                      #
```

```
9  # mpiexec -np <cantidad_procesos> python3 mpi.py <matriz_A> <matriz_B>        #
10 # -np<num_prueba> -cn<num_nodos> -pp<num_procesos_por_nodo>                    #
11 # Ejemplo:                                                                    #
12 # mpiexec -np 4 python3 mpi.py 4x4 4x4 -np1                                   #
13 # mpiexec -np 4 python3 mpi.py 4x4 4x4 -np1 -cn2 -pp2                         #
14 ###############################################################################
15 #                                 Librerías                                   #
16 ###############################################################################
17 import numpy as np          # Librería para manejo de matrices.
18 from mpi4py import MPI      # Librería para manejo de MPI.
19 import socket               # Librería para manejo de hostname.
20 import time                 # Librería para manejo de tiempo.
21 import sys                  # Librería para manejo de argumentos.
22 import json                 # Librería para manejo de archivos json.
23 import logging              # Librería para manejo de logs.
24 ###############################################################################
25 #                                 Constantes                                  #
26 ###############################################################################
27 DIR_LOG = '/clusterfs/docs/logs/ethernet/' # Directorio para los logs.
28 ###############################################################################
29 #                                Funciones                                    #
30 ###############################################################################
31 def actualizar_json(nombre='', datos={}):
32     """ Función para actualizar un archivo json.
33     Argumentos:
34         nombre {str} -- Es el nombre del archivo json. (default: {''})
35         datos {dict} -- Es el diccionario con los datos a guardar.
36         (default: {{}})
37     """
38     # Se recupera la información del archivo json, si no existe se crea.
39     try:
40         with open(nombre, 'r') as archivo:
41             info = json.load(archivo)
42     except:
43         info = {}
44     # Se actualiza la información del archivo json.
45     info.update(datos)
46     # Se escribe la información en el archivo json.
47     with open(nombre, 'w') as archivo:
48         json.dump(info, archivo, indent=4)
49     return
50 def configurar_loggers(nombre='', ruta=''):
51     """Esta función configura los loggers para escribir en un archivo.
52     Argumentos:
53         nombre {str} -- Es el nombre del logger. (default: {''})
54         ruta {str} -- Es la ruta del archivo de log. (default: {''})
55     Retorna:
56         {logger} -- Es el logger configurado.
57     """
58     # Se crea el logger.
59     logger = logging.getLogger(nombre)
60     # Se crea el handler para escribir en un archivo.
61     handler = logging.FileHandler(ruta)
62     # Se crea el formato del mensaje.
63     formato_mensaje = '%(asctime)s|%(name)s(%(levelname)s):%(message)s'
64     formatter = logging.Formatter(formato_mensaje)
65     # Se agrega el formato al handler.
66     handler.setFormatter(formatter)
67     # Se agrega el handler al logger.
68     logger.addHandler(handler)
69     # Se configura el nivel del logger.
70     logger.setLevel(logging.DEBUG)
71     return logger
72 def main():
73     # Se obtiene el tiempo inicial.
74     tiempo_inicial = time.time()
75     # Se obtiene el hostname.
76     hostname = socket.gethostname()
```

```
77      # Se crea el comunicador.
78      comm = MPI.COMM_WORLD
79      # Se obtiene el tamaño del cluster.
80      cantidad_procesos = comm.Get_size()
81      # Se obtiene el rango del nodo (o ID del proceso).
82      rango = comm.Get_rank()
83      # Se inicializa el diccionario para guardar los datos.
84      datos = {}
85      datos_salida = {'Tiempo_total': 0.0, 'Error': ""}
86      # Se guarda el hostname.
87      datos['hostname'] = hostname
88      # Se guarda el rango.
89      datos['rango'] = rango
90      # Se guarda la cantidad de procesos.
91      datos['total_procesos'] = cantidad_procesos
92      datos['nodos'] = ""
93      datos['proceso_por_nodo'] = ""
94      # Se genera el nombre del archivo de log.
95      archivo_log = f"{DIR_LOG}{hostname}_rank_{rango}.log"
96      # Se genera el nombre del archivo json.
97      archivo_json = f"{DIR_LOG}{hostname}_rank_{rango}.json"
98      # Se configura el logger.
99      logger = configurar_loggers(f"{hostname}_rank_{rango}", archivo_log)
100     try:
101         # Se obtienen las dimensiones de las matrices.
102         dimension = sys.argv[1].split('x') # Se separan las dimensiones.
103         filas_A = int(dimension[0])   # Se obtienen las filas de la matriz A.
104         columnas_A = int(dimension[1])# Se obtienen las columnas de la matriz A.
105         dimension = sys.argv[2].split('x') # Se separan las dimensiones.
106         filas_B = int(dimension[0])   # Se obtienen las filas de la matriz B.
107         columnas_B = int(dimension[1])# Se obtienen las columnas de la matriz B.
108         # Se revisa si se adjuntó el número de prueba.
109         numero_prueba = '1'
110         # Se revisa si existen más de 3 argumentos.
111         if len(sys.argv) >3:
112             # Se revisa cada uno de los argumentos restantes.
113             for i in range(3, len(sys.argv)):
114                 # Se revisa si el argumento es el número de prueba.
115                 if sys.argv[i][:3] == '-np':
116                     # Se obtiene el número de prueba.
117                     numero_prueba = sys.argv[i][3:]
118                 # Se revisa si el argumento es la cantidad de nodos.
119                 if sys.argv[i][:3] == '-cn':
120                     # Se obtiene la cantidad de nodos.
121                     datos['nodos'] = sys.argv[i][3:]
122                 # Se revisa si el argumento es la cantidad de procesos por nodo.
123                 if sys.argv[i][:3] == '-pp':
124                     # Se obtiene la cantidad de procesos por nodo.
125                     datos['proceso_por_nodo'] = sys.argv[i][3:]
126         # Se genera la llave para el archivo json.
127         llave = f"Prueba_{numero_prueba}_{filas_A}x{columnas_A}"
128         llave += f"_{filas_B}x{columnas_B}"
129         # Se realiza la validación de las dimensiones de las matrices.
130         if columnas_A != filas_B:
131             logger.error('Las matrices no se pueden multiplicar.')
132             print('Las matrices no se pueden multiplicar.')
133             # Se guarda el error en el archivo json.
134             datos_salida['Error'] = 'Las matrices no se pueden multiplicar.'
135             datos['Salida']= datos_salida
136             # Se actualiza el archivo json.
137             actualizar_json(archivo_json, {llave: datos})
138             return
139         # Se realiza la validación de la cantidad de nodos.
140         if cantidad_procesos > filas_A:
141             # Si el número de procesos es mayor a la cantidad de filas de la
142             # matriz A, se ajusta la cantidad de procesos.
143             cantidad_procesos = filas_A
144             # Se revisa si el rango actual es mayor a la cantidad de procesos.
```

```
145             if (rango+1) >= cantidad_procesos:
146                 # Si el rango es mayor a la cantidad de procesos, se termina el
147                 # proceso.
148                 msg_info = f"Se termina el proceso {rango}."
149                 msg_info += " Debido a que la cantidad de procesos es mayor "
150                 msg_info += "a la cantidad de filas de la matriz A."
151                 logger.info(msg_info)
152                 # Se guarda el error en el archivo json.
153                 datos_salida['Error'] = "Proceso no necesario."
154                 datos['Salida']= datos_salida
155                 # Se actualiza el archivo json.
156                 actualizar_json(archivo_json, {llave: datos})
157                 return
158         # Se obtiene la cantidad de filas por proceso.
159         filas_por_proceso = filas_A // cantidad_procesos
160         # Se obtiene la cantidad de filas restantes.
161         filas_restantes = filas_A % cantidad_procesos
162         # Se ajusta filas_por_proceso si este proceso va a recibir las filas
163         # restantes
164         if rango < filas_restantes:
165             filas_por_proceso += 1
166         # Se genera la matrices A y B.
167         if rango == 0:
168             # Se genera la matriz A.
169             matriz_A = 100*np.random.rand(filas_A, columnas_A)
170             matriz_A.astype(np.float)
171             # Se muestra la matriz A.
172             msg_info = f"Matriz A:"
173             msg_info += f"\nTamaño {matriz_A.shape[0]}x{matriz_A.shape[1]}"
174             msg_info += f"\n{matriz_A}"
175             # Se genera la matriz B.
176             matriz_B = 100*np.random.rand(filas_B, columnas_B)
177             matriz_B.astype(np.float)
178             # Se muestra la matriz B.
179             msg_info += f"\nMatriz B:"
180             msg_info += f"\nTamaño {matriz_B.shape[0]}x{matriz_B.shape[1]}"
181             msg_info += f"\n{matriz_B}"
182             # Se muestra la información en el archivo log.
183             logger.info(msg_info)
184         else:
185             # Se generan las matrices A y B vacías.
186             matriz_A = np.empty((filas_por_proceso, columnas_A), dtype=np.float)
187             matriz_B = np.empty((filas_B, columnas_B), dtype=np.float)
188         # Se envía la matriz B a todos los procesos.
189         matriz_B = comm.bcast(matriz_B, root=0)
190         # Se envían las filas de la matriz A correspondientes a cada proceso.
191         matriz_A_local= np.empty((filas_por_proceso,columnas_A), dtype=np.float)
192         # Se envían las filas de la matriz A.
193         comm.Scatter(matriz_A, matriz_A_local, root=0)
194         # Se muestran las matrices A y B.
195         msg_info = f"Matriz A de rank {rango} para cálculos:"
196         msg_info += f"\nTamaño {matriz_A_local.shape[0]}x"
197         msg_info += f"{matriz_A_local.shape[1]}\n{matriz_A_local}"
198         logger.info(msg_info)
199         msg_info = f"Matriz B de rank {rango} para cálculos:"
200         msg_info += f"\nTamaño {matriz_B.shape[0]}x{matriz_B.shape[1]}"
201         msg_info += f"\n{matriz_B}"
202         logger.info(msg_info)
203         # Se realiza la multiplicación de las matrices.
204         matriz_C_local = np.matmul(matriz_A_local, matriz_B)
205         # Se muestra la matriz resultante.
206         msg_info = f"Rank {rango}, Resultado Local:"
207         msg_info += f"\nTamaño {matriz_C_local.shape[0]}x"
208         msg_info += f"{matriz_C_local.shape[1]}\n{matriz_C_local}"
209         logger.info(msg_info)
210         # Se recopilan los resultados de todos los procesos en el proceso 0.
211         # Todos los procesos deben tener una matriz result inicializada.
212         matriz_C = np.empty((filas_A, columnas_B), dtype=np.float)
```

```
213         comm.Gather(matriz_C_local, matriz_C, root=0)
214         # Si es el proceso 0, se muestra las matriz resultante.
215         if rango == 0:
216             msg_info = f"Matriz resultante:"
217             msg_info += f"\nTamaño {matriz_C.shape[0]}x{matriz_C.shape[1]}"
218             msg_info += f"\n{matriz_C}"
219             logger.info(msg_info)
220         # Se obtiene el tiempo final.
221         tiempo_final = time.time()
222         # Se obtiene el tiempo total.
223         tiempo_total = tiempo_final - tiempo_inicial
224         # Se guarda el tiempo total.
225         datos_salida['Tiempo_total'] = tiempo_total
226         # Se guarda la salida.
227         datos['Salida']= datos_salida
228         # Se actualiza el archivo json.
229         actualizar_json(archivo_json, {llave: datos})
230         # Se muestra la información en el archivo log.
231         msg_info = f"Tiempo total: {tiempo_total} segundos."
232         logger.info(msg_info)
233         MPI.Finalize() # Se finaliza el proceso.
234     except Exception as e: # Se captura cualquier excepción.
235         logger.error(e) # Se muestra el error en el archivo log.
236         MPI.Finalize() # Se finaliza el proceso.
237         sys.exit(1)
238     return
239 ################################################################################
240 if __name__ == '__main__':
241     main()
```

Cuadro 35: Código de multiplicación de matrices.

CAPÍTULO 8

Configuración de *cluster* I2C

En la Figura 19 se puede observar un diagrama con la topología del *cluster* I2C, la cual puede expandirse hasta 112 nodos esclavos. La topología se implementó con un maestro y 2 esclavos, los cuales se se encargan de realizar una multiplicación de matrices. El maestro se encarga de enviar la información de las dimensiones de las matrices, la matriz A y la parte correspondiente de la matriz B a cada nodo esclavo. Los nodos esclavos se encargan de recibir la información enviada por el nodo maestro, realizar el cálculo de su parte de la matriz C y enviar la parte de la matriz C resultante al nodo maestro. El nodo maestro se encarga de recibir las partes de la matriz C enviadas por los nodos esclavos y unirlas en una sola matriz C.

Como se puede apreciar en el diagrama de la topología, el nodo maestro utiliza los pines de propósito general (GPIO) 2 y 3 para los buses de información y reloj, respectivamente. Mientras que los nodos esclavos utilizan los pines de propósito general 18, para el bus de información, y 19, para el reloj. La diferencia en los pines utilizados por el nodo maestro y los nodos esclavos se debe a que estos últimos al configurarse como esclavos I2C, el procesador de la Raspberry Pi asigna esos pines.

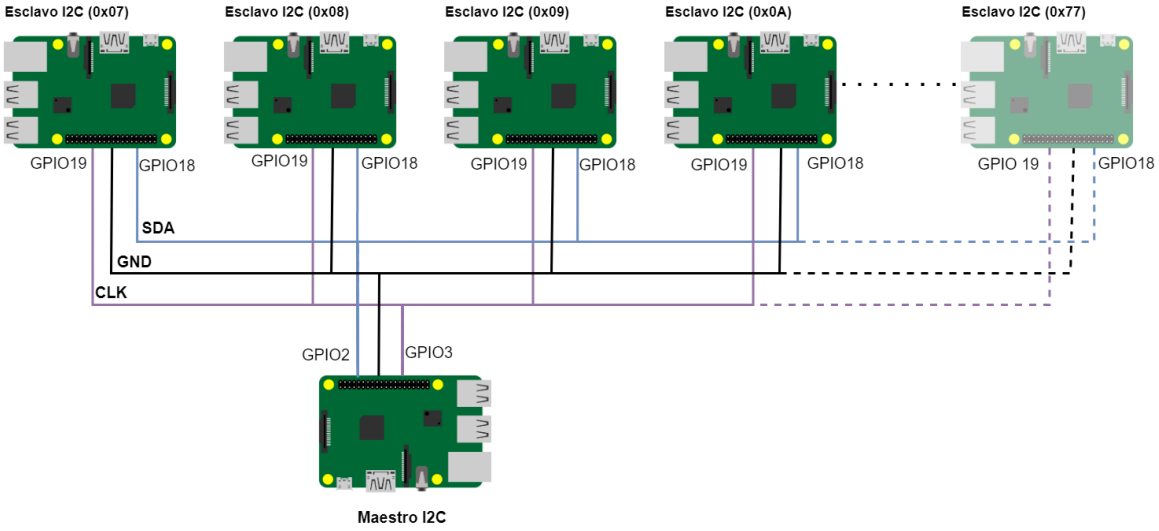


Figura 19: Diagrama del *cluster* I2C.

8.1. Código de nodo maestro

El código de lenguaje Python del Cuadro 36 se diseñó para configurar a la Raspberry Pi como un maestro de I2C. Al momento de ejecutar el código, este primero revisa que las dimensiones de las matrices permitan la realización de la multiplicación de matrices, posteriormente revisa que la cantidad de bytes totales por matrices no superen los 255 bytes, debido a que el código genera una serie de paquetes los cuales en el encabezado dedica únicamente 1 byte para indicar la cantidad de bytes totales de datos que contendrán las matrices, por lo que si se superan los 255 bytes de datos, el código no funcionará de manera correcta.

Una vez que se verifica lo anterior, el código procede a generar las matrices A y B, para después separar la matriz B en partes, con igual cantidad de columnas para cada nodo esclavo. En caso el tamaño de matrices no sea divisible entre la cantidad de nodos totales (maestro más esclavos), el nodo maestro se queda con todas las columnas sobrantes. Otro aspecto que toma en cuenta al momento de separar la matriz B, es cuando la cantidad de nodos totales es mayor a la cantidad de columnas de la matriz B, en este caso se eliminan los nodos esclavos sobrantes.

Posteriormente, el código se encarga de iniciar la comunicación enviando primero la información de las dimensiones de las matrices a cada esclavo, luego procede a transmitir la información de la matriz A a todos los esclavos y, por último, la parte correspondiente de la matriz B a cada nodo esclavo. Una vez que los nodos esclavos han recibido la información, el código procede a realizar el cálculo de su parte de la matriz C, para después recolectar las demás partes de la matriz C enviadas por los nodos esclavos y unirlas en una sola matriz C. Finalmente, el código se encarga de imprimir la matriz C resultante y el tiempo total de ejecución del código. En la Figura 20 se puede observar un diagrama de flujo del código del nodo maestro.

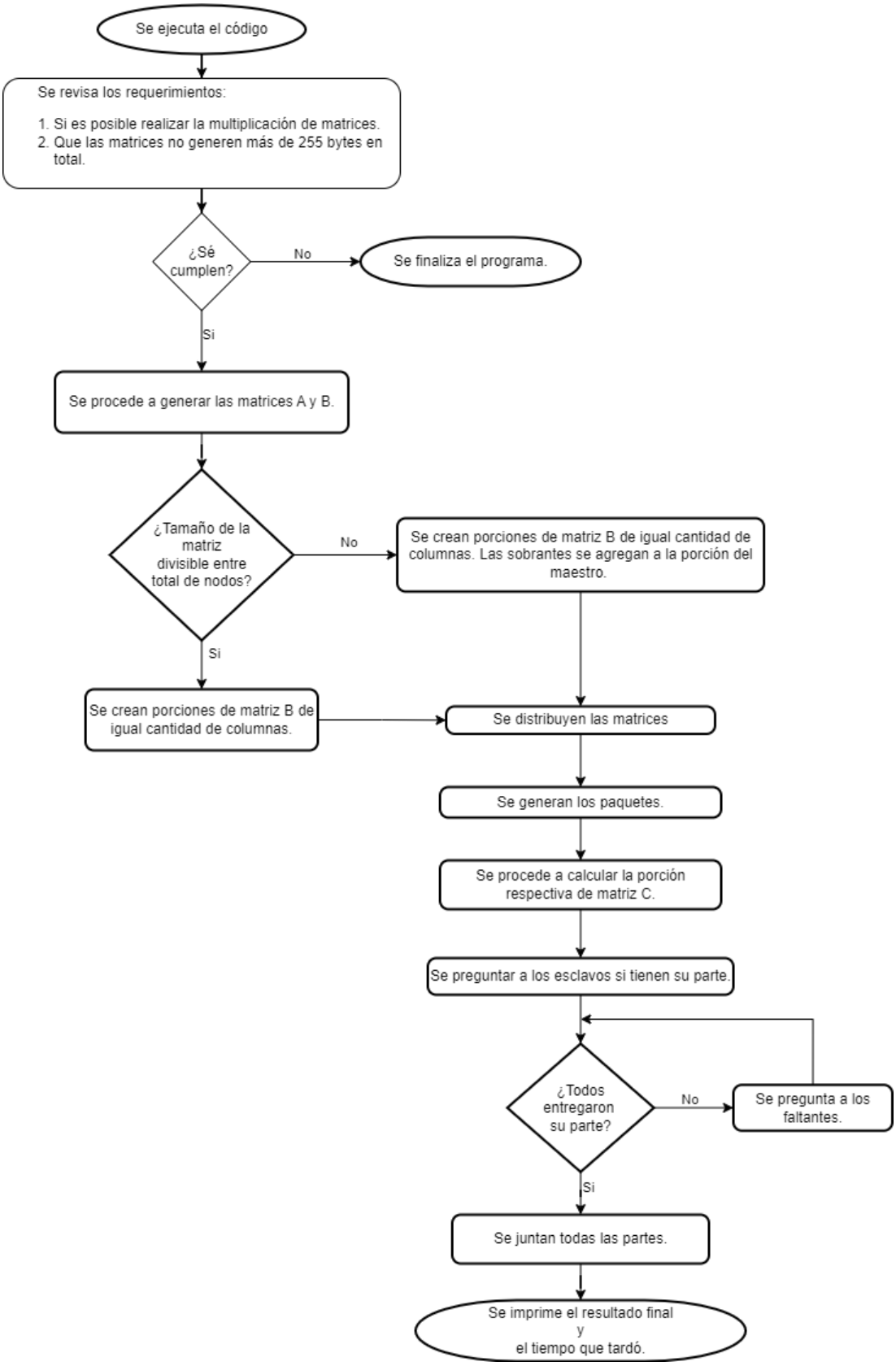


Figura 20: Diagrama de flujo del código del nodo maestro.

```
##############################################################################
# Autor: Daniel Mundo                                                        #
# Fecha: 2024-01-06                                                          #
# Descripción:                                                               #
#   Programa para realizar la multiplicación de matrices utilizando el       #
#   protocolo de comunicación I2C.                                           #
#   El programa se ejecuta de la siguiente manera:                           #
#   python3 i2c_master.py <filas_A>x<columnas_A> <filas_B>x<columnas_B>      #
#   <direcciones_esclavos>                                                   #
#   - filas_A: Número de filas de la matriz A.                               #
#   - columnas_A: Número de columnas de la matriz A.                         #
#   - filas_B: Número de filas de la matriz B.                               #
#   - columnas_B: Número de columnas de la matriz B.                         #
#   - direcciones_esclavos: Lista de direcciones de los esclavos separados   #
#                           por comas, este puede ser opcional.              #
##############################################################################
#                                Librerías                                   #
##############################################################################
import traceback        # Librería para mostrar los errores.
import pigpio           # Librería para controlar los pines de la Raspberry Pi.
import time             # Librería para gestionar el tiempo.
import sys              # Librería para gestionar los argumentos de entrada.
import numpy as np      # Librería para manejo de matrices.
import logging          # Librería para gestionar los logs.
##############################################################################
#                                Constantes                                  #
##############################################################################
# Configuración (mismo para maestro y esclavo).
IS_ACK_AVAILABLE = True # Bandera para indicar si se debe esperar el ACK.
PACKET_DATA_SIZE = 13   # Tamaño de los paquetes de datos.
PACKET_HEAD_SIZE = 2    # Tamaño de los encabezados de los paquetes.
PACKET_TAIL_SIZE = 1    # Tamaño de las colas de los paquetes.
SDA_PIN          = 2    # Pin de bus de datos.
SCL_PIN          = 3    # Pin de reloj.
# Direcciones de los registros (mismo para maestro y esclavo).
REGISTER_REQUEST = 0x01 # Dirección del registro para solicitar información.
REGISTER_SIZE    = 0x02 # Dirección del registro de tamaño de matrices.
REGISTER_DATA_A  = 0x04 # Dirección del registro de datos de la matriz A.
REGISTER_DATA_B  = 0x08 # Dirección del registro de datos de la matriz B.
# Estados de la comunicación (mismo para maestro y esclavo).
START_COMM      = 0xF0  # Inicio de envío de información.
END_COMM        = 0x0F  # Finalización de la comunicación.
END_PACKET      = 0x10  # Finalización del paquete de información.
START_OVERFLOW  = 0x20  # Inicio de la comunicación con desbordamiento.
OVERFLOW_DATA   = 0x40  # Bandera para indicar que hay desbordamiento de datos.
# Arreglos con las direcciones de los esclavos.
SLAVE_ADDRESS   = [0x47]
##############################################################################
#                               Funciones                                    #
##############################################################################
def checksum(data= bytearray()):
    """
    Calcula el checksum de un arreglo de bytes utilizando un algoritmo
    personalizado.

    Parámetros:
        data {bytearray}: Arreglo de bytes a partir del cual se calculará el
        checksum. (default: {bytearray()})

    Retorna:
        checksum {bytes}: Checksum calculado.
    """
    checksum = 0
    for i, byte in enumerate(data):
        checksum ^= byte
        # Se rota el checksum 2 bits a la derecha si el índice es par, caso
        # contrario se rota 2 bits a la izquierda.
```

```
68          if(i%2 == 0): checksum ^= byte << 2
69          else: checksum ^= byte >> 2
70      # Se rota el checksum 1 bit a la derecha.
71      checksum = (checksum >> 1) & 0xFF
72      # Se convierte el checksum a bytes.
73      checksum = checksum.to_bytes(1, byteorder='big')
74      return checksum # Se retorna el checksum calculado como byte.
75  def generate_data_packets(register=int(), data=bytes()):
76      """En esta función se genera los paquetes de datos a enviar por el bus I2C.
77
78      Argumentos:
79          register {int} -- La variable contiene el registro y el estado de la
80          comunicación. (default: {int()})
81
82          data {bytes} -- Arreglo de bytes a enviar por el bus I2C.
83          (default: {bytes()})
84
85      Retorna:
86          packets {list} -- Lista de paquetes de datos a enviar por el bus I2C.
87      """
88      num_bytes   = len(data)
89      # Se calcula el número de paquetes de datos a generar.
90      num_packets = num_bytes//PACKET_DATA_SIZE
91      res_packets = num_bytes%PACKET_DATA_SIZE
92      # Se crea una lista vacía para almacenar los paquetes de datos.
93      packets = []
94          # Se genera el paquete del arreglo de bytes.
95      if (abs(num_packets) < 1) or ((num_packets == 1) and (res_packets == 0)):
96          if IS_ACK_AVAILABLE:
97              # Se calcula el checksum.
98              local_checksum = checksum(bytearray(data))
99          else:
100             local_checksum = b''
101         # data_to_send = Tamaño de datos + datos + checksum.
102         data_to_send =  bytes([num_bytes])+ data + local_checksum
103         # Se genera el paquete y se agrega a la lista de paquetes.
104         packets = [[int(END_PACKET|register),             # Estado y registro.
105                     bytes(data_to_send)]]                 # Campo de data.
106     else:
107         # Se revisa que no existan bytes sobrantes, si hay se agrega otro
108         # paquete más.
109         if res_packets !=0 : num_packets +=1
110         # Se calcula el checksum.
111         if IS_ACK_AVAILABLE:
112             local_checksum = checksum(bytearray(data[:PACKET_DATA_SIZE]))
113         else:
114             local_checksum = b''
115         # Se genera el campo de data.
116         data_to_send = bytes([num_bytes])+data[:PACKET_DATA_SIZE]+\
117             local_checksum
118         # Se genera el primer paquete.
119         packets = [[int(START_OVERFLOW|register),          # Estado y registro.
120                     bytes(data_to_send)]]                 # Campo de data.
121         # Se genera el/los paquete(s) de la matriz A.
122         for i in range(0, num_packets-2):
123             # Calculando el checksum.
124             if IS_ACK_AVAILABLE:
125                 # Variable auxiliar para convertir de bytes a bytearray.
126                 aux_b = bytearray(
127                     data[PACKET_DATA_SIZE*(i+1):PACKET_DATA_SIZE*(i+2)]
128                                 )
129                 local_checksum = checksum(aux_b)
130             else:
131                 local_checksum = b''
132             # Se genera el campo de data.
133             data_to_send = bytes([num_bytes])+\
134                 data[PACKET_DATA_SIZE*(i+1):PACKET_DATA_SIZE*(i+2)]+\
135                     local_checksum
```

```
136             # Se genera el paquete y se agrega a la lista de paquetes.
137             packets.append([int(OVERFLOW_DATA|register),# Estado y registro.
138                             bytes(data_to_send)         # Campo de data.
139                             ])
140         # Se genera el último paquete.
141         # Calculando el checksum.
142         if IS_ACK_AVAILABLE:
143             # Variable auxiliar para convertir de bytes a bytearray.
144             aux_b = bytearray(
145                 data[PACKET_DATA_SIZE*(num_packets-1):]
146                                 )
147             local_checksum = checksum(aux_b)
148         else:
149             local_checksum = b''
150         # Se genera el campo de data.
151         data_to_send = bytes([num_bytes])+\
152             data[PACKET_DATA_SIZE*(num_packets-1):]+local_checksum
153 
154         # Se genera el paquete y se agrega a la lista de paquetes.
155         packets.append([int(END_PACKET|register),         # Estado y registro.
156                         bytes(data_to_send)])             # Campo de data.
157     return packets # Se retorna la lista de paquetes de datos.
158 def wait_for_ack(value_to_check = bytes(), slave= int(), pi= pigpio.pi(),
159                  logs= logging.getLogger(), timeout= 180.0):
160     """
161     Espera a que se reciba un ACK (acknowledge) del dispositivo esclavo en
162     el bus I2C.
163 
164     Parámetros:
165         - slave: Dirección del dispositivo esclavo.
166         - pi: Instancia de la clase pigpio para controlar el bus I2C.
167         - timeout: Tiempo máximo de espera para recibir el ACK.
168         - logs: Instancia de la clase logging para registrar eventos.
169 
170     Retorna:
171         - ack: True si se recibió el ACK, False si no se recibió.
172     """
173     # Se define el tiempo inicial.
174     start_time = time.time()
175     # Se define el tiempo actual.
176     current_time = start_time
177     # Se define el estado del ACK.
178     ack = False
179     logs.debug(f"Valor esperado de ACK {value_to_check}")
180     # Se espera a que se reciba el ACK.
181     while (current_time - start_time) < timeout:
182         # Se pregunta si se ha recibido el ACK.
183         data = pi.i2c_read_byte(slave)
184         logs.debug(f"Valor recibido de ACK {bytes([data])}")
185         # Si se ha recibido el ACK, se sale del ciclo.
186         if bytes([data]) == value_to_check:
187             logs.debug("Se recibió el ACK")
188             ack = True
189             break
190         time.sleep(1)
191         # Se obtiene el tiempo actual.
192         current_time = time.time()
193     # Se muestra mensaje de error si no se recibió el ACK.
194     if ack == False:
195         logs.debug("No se recibió el ACK")
196     return ack # Se retorna el estado del ACK.
197 def send_data_packets(slave= int(), packet_size= list(),
198                       packet_matrix_A= list(), packet_matrix_B= list(),
199                       pi= pigpio.pi(), log= logging.getLogger()):
200     """
201     Función para enviar los paquetes de datos a los esclavos.
202 
203     Parámetros:
```

```
204         - slave: Dirección del esclavo al que se le enviará la información.
205         - packet_size: Paquete de datos con el tamaño de las matrices.
206         - packet_matrix_A: Paquete de datos con la matriz A.
207         - packet_matrix_B: Paquete de datos con la matriz B.
208         - pi: Instancia de la clase pigpio para controlar el bus I2C.
209         - log: Instancia de la clase logging para registrar eventos.
210 
211     Retorna:
212         - Tiempo que se demoró en enviar los paquetes de datos.
213     """
214     log.debug(f"Se envia información al Esclavo_{hex(slave)}")
215     # Se define la lista de paquetes de datos.
216     packets = []
217     #* Se revisa si packet_size es una lista de listas. *#
218     if isinstance(packet_size[0], list):
219         # Se agrega el paquete de dimensionales de las matrices.
220         packets.extend(packet_size)
221     else:
222         # Se agrega el paquete de dimensionales de las matrices.
223         packets.append(packet_size)
224     log.debug(f"Paquetes de tamaño:\n{packet_size}")
225     #* Se revisa si packet_matrix_A es una lista de listas. *#
226     if isinstance(packet_matrix_A[0], list):
227         # Se agrega el paquete de la matriz A.
228         packets.extend(packet_matrix_A)
229     else:
230         # Se agrega el paquete de la matriz A.
231         packets.append(packet_matrix_A)
232     #* Se revisa si packet_matrix_B es una lista de listas. *#
233     if isinstance(packet_matrix_B[0], list):
234         # Se agrega el paquete de la matriz B.
235         packets.extend(packet_matrix_B)
236     else:
237         # Se agrega el paquete de la matriz B.
238         packets.append(packet_matrix_B)
239     # Se define el tiempo inicial de envío de información.
240     local_start_time = time.time()
241     # Se envía el paquete de inicio de comunicación.
242     pi.i2c_write_byte(slave, START_COMM)
243     time.sleep(1)
244     # Se procede a enviar cada paquete de datos.
245     for packet in packets:
246         msg_debug =f"Enviando Paquete:\nRegistro:{hex(packet[0])}"
247         msg_debug+=f"\nData:{packet[1]}"
248         log.debug(msg_debug)
249         # Se envía el paquete de datos.
250         pi.i2c_write_i2c_block_data(slave, packet[0], packet[1])
251         # Se espera a que se reciba el ACK.
252         if IS_ACK_AVAILABLE:
253             # Se recupera el checksum del paquete de datos.
254             checksum_calculated = packet[1][-1:]
255             wait_for_ack(checksum_calculated, slave, pi, logs=log)
256         time.sleep(0.2)
257     # Se envía el paquete de finalización de la comunicación.
258     pi.i2c_write_byte(slave, END_COMM)
259     time.sleep(0.15)
260     # Se define el tiempo final.
261     local_end_time = time.time() - local_start_time
262     return local_end_time # Se retorna el tiempo de envío de información.
263 def receive_data_packets(slave=int(), pi=pigpio.pi(), log=logging.getLogger()):
264     """
265     Función para recibir los paquetes de datos de los esclavos.
266 
267     Parámetros:
268         - slave: Dirección del esclavo del que se recibirá la información.
269         - pi: Instancia de la clase pigpio para controlar el bus I2C.
270         - log: Instancia de la clase logging para registrar eventos.
271 
```

```
272     Retorna:
273         - matrix_c_data: Bytes de la matriz C recibida del esclavo.
274     """
275     # Se define el registro a enviar data.
276     register = END_PACKET|REGISTER_REQUEST
277     # Se procede a enviar el paquete para preguntar por la información.
278     log.debug(f"Se solicita información al Esclavo_{hex(slave)}")
279     pi.i2c_write_byte(slave, register)
280     time.sleep(1)
281     # Se inicializa la variable para almacenar la información recibida.
282     list_matrix_c_data = []
283     matrix_c_data = b''
284     packet_total_size  = PACKET_HEAD_SIZE+PACKET_DATA_SIZE
285     packet_total_size += (IS_ACK_AVAILABLE*PACKET_TAIL_SIZE)
286     while True:
287         # Se procede a leer la información disponible.
288         n, data_bytes = pi.i2c_read_i2c_block_data(slave, 0x00,
289                                                    packet_total_size)
290         if n == packet_total_size:
291             # Se convierte data_bytes en tipo bytes.
292             if isinstance(data_bytes, bytearray):
293                 data_bytes = bytes(data_bytes)
294             else:
295                 data_bytes = bytes(data_bytes, 'utf-8')
296             int_register = int.from_bytes(data_bytes[:1], byteorder='big')
297             number_bytes = int.from_bytes(data_bytes[1:2], byteorder='big')
298             # Se revisa que el paquete recibido sea de la acción solicitada.
299             if (int_register & 0x0F) == REGISTER_REQUEST:
300                 # Se muestra mensaje de debug.
301                 log.debug(f"Se recibió:\n{data_bytes}")
302                 # Se procede a revisar si la información recibida es del
303                 # registro END_PACKET y no contiene ceros posteriores al
304                 # checksum.
305                 if (int_register & 0xF0) == END_PACKET:
306                     if number_bytes < PACKET_DATA_SIZE:
307                         data_bytes = data_bytes[:number_bytes+3]
308                         log.debug(f"Data ajustada:\n{data_bytes}")
309                 # Se calcula el checksum.
310                 if IS_ACK_AVAILABLE:
311                     checksum_calculated = checksum(
312                         bytearray(data_bytes[2:-1])
313                         )
314                     # Se muestra el checksum calculado y el checksum del
315                     # paquete.
316                     msg_debug = f"Checksum calculado: {checksum_calculated}"
317                     msg_debug += f"\nChecksum recibido: {data_bytes[-1:]}"
318                     log.debug(msg_debug)
319                     # Se compara el checksum calculado con el recibido.
320                     if checksum_calculated == data_bytes[-1:]:
321                         # Se envía un ACK.
322                         pi.i2c_write_byte(slave,
323                                           int.from_bytes(checksum_calculated,
324                                                          byteorder='big'))
325                         list_matrix_c_data.append(data_bytes[2:-1])
326                         log.debug(f"Información actual de la lista:
                                {list_matrix_c_data}")
327                     else:
328                         # Se envía un NACK.
329                         pi.i2c_write_byte(slave,
330                                           int.from_bytes(checksum_calculated,
331                                                          byteorder='big'))
332                         # Se muestra mensaje de error.
333                         log.debug("Error en el checksum")
334                 else:
335                     list_matrix_c_data.append(data_bytes[2:-1])
336                     log.debug(f"Información actual de la lista:
                            {list_matrix_c_data}")
337             else:
```

```
338                 # Se muestra mensaje de debug.
339                 log.debug(f"Esclavo {slave}: No tiene información disponible")
340             if data_bytes[0] == END_PACKET|REGISTER_REQUEST:
341                 # Se muestra mensaje de debug.
342                 log.debug(f"Esclavo {slave}: Finalización de la comunicación")
343                 # Se concatena toda los bytes recibidos.
344                 matrix_c_data = matrix_c_data.join(list_matrix_c_data)
345                 break
346         time.sleep(0.5)
347     return matrix_c_data
348 def setup_logger(name='', log_file=''):
349     """
350     Función para configuración del log que contiene la información de la
351     ejecución del programa.
352 
353     Parámetros:
354         - name: Nombre del archivo de logs.
355         - log_file: Nombre del archivo de logs.
356 
357     Retorna:
358         - logger: Objeto de la clase logging.
359     """
360     # Se crea el formato de los logs.
361     formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
362     # Se crea el archivo de logs.
363     handler = logging.FileHandler(log_file)
364     # Se agrega el formato al archivo de logs.
365     handler.setFormatter(formatter)
366     # Se crea el objeto de logs.
367     logger = logging.getLogger(name)
368     # Configurando el nivel de logs.
369     logger.setLevel(logging.INFO)
370     # Se agrega el formato al objeto de logs.
371     logger.addHandler(handler)
372     return logger # Se retorna el objeto de logs.
373 def main():
374     # Se obtiene el tiempo inicial.
375     start_time = time.time()
376     # Se inicializa la variable que almacena los errores.
377     msg_error = ''
378     # ---------------------------------------------------------------------
379     # * Definiendo tamaño de la matriz. *
380     try:
381         # Se obtienen las dimensiones de la matriz.
382         dim = sys.argv[1].split('x')
383         num_rows_A = int(dim[0]) # Se define número de filas de la matriz A.
384         num_cols_A = int(dim[1]) # Se define número de columnas de la matriz A.
385         dim = sys.argv[2].split('x')
386         num_rows_B = int(dim[0]) # Se define número de filas de la matriz B.
387         num_cols_B = int(dim[1]) # Se define número de columnas de la matriz B.
388     # Se captura la excepción si no se ingresaron los argumentos de entrada.
389     except IndexError:
390         # Se muestra mensaje de error.
391         msg_error += "No se ingresaron los argumentos de entrada de manera "
392         msg_error += "correcta.\nLos argumentos de entrada deben ser de la "
393         msg_error += "forma: python3 i2c_master.py <filas_A>x<columnas_A> "
394         msg_error += "<filas_B>x<columnas_B>\n"
395         print(msg_error)
396         # Se termina la ejecución del programa.
397         sys.exit(1)
398     # Se revisa si existe un tercer argumento de entrada.
399     if len(sys.argv) > 3:
400         # Se recupera el tercer argumento de entrada, este es la lista de
401         # direcciones de los esclavos.
402         address = sys.argv[3].split(',')
403         # Se reemplaza la lista de direcciones de los esclavos actual, por
404         # la nueva y se convierte en una variable global.
405         global SLAVE_ADDRESS
```

```
406         SLAVE_ADDRESS = [int(addr, 16) for addr in address]
407     # Se obtiene la cantidad de bytes totales de las matrices.
408     limit_byte_matrix_A = 4 * num_rows_A * num_cols_A
409     limit_byte_matrix_B = 4 * num_rows_B * num_cols_B
410     # Se comprueba si es posible realizar la multiplicación de matrices.
411     if not (num_cols_A == num_rows_B):
412         # Se muestra un mensaje de error.
413         msg_error += "No es posible realizar la multiplicación de matrices,"
414         msg_error += " debido a que las columnas de la matriz A no es igual"
415         msg_error += " a las filas de la matriz B:"
416         msg_error += f"\n-Matriz A: {sys.argv[1]}\n-Matriz B: {sys.argv[2]}\n"
417     # Se comprueba si la cantidad de bytes totales de las matrices es mayor a
418     # 255 bytes.
419     if (limit_byte_matrix_A | limit_byte_matrix_B) > 255:
420         # Se muestra mensaje de error.
421         msg_error += "La cantidad de bytes totales por matriz es mayor de"
422         msg_error += " 255 bytes."
423         msg_error += f"\nTotal de bytes Matriz A: 4x{sys.argv[1]}="
424         msg_error += f"{4*num_rows_A*num_cols_A}"
425         msg_error += f"\nTotal de bytes Matriz B: 4x{sys.argv[2]}="
426         msg_error += f"{4*num_rows_B*num_cols_B}"
427     # ------------------------------------------------------------------
428     # * Se configura el Log. *
429     # Se obtiene el nombre del archivo de python actual.
430     file_name = sys.argv[0].split('/')[-1]
431     # Se define el nombre del archivo de Logs.
432     log_file  = "/clusterfs/docs/logs/i2c/"
433     log_file += f"{file_name[:-3]}_MA{sys.argv[1]}_MB{sys.argv[2]}.log"
434     log = setup_logger('i2c_master', log_file)
435     # Se comprueba si existieron errores.
436     if msg_error != '':
437         # Se muestra mensaje de error.
438         log.error(msg_error)
439         # Se termina la ejecución del programa.
440         sys.exit(1)
441     # ------------------------------------------------------------------
442     #* Se generan las matrices a enviar vía I2C. *
443     matrix_A = 100*np.random.rand(num_rows_A, num_cols_A)
444     matrix_B_raw = 100*np.random.rand(num_rows_B, num_cols_B)
445     # Se convierte las matrices en matrices float 32.
446     matrix_A = matrix_A.astype(np.float32)
447     matrix_B_raw = matrix_B_raw.astype(np.float32)
448     # ------------------------------------------------------------------
449     #* Se define la cantidad de dispositivos esclavos.
450     num_nodes = len(SLAVE_ADDRESS)+1
451     # TODO: Se comprueba el tamaño de la matriz B y la cantidad de nodos.
452     if num_nodes > num_cols_B:
453         # Si el número de nodos es mayor al número de columnas de la matriz
454         # significa que es necesario eliminar nodos.
455         # Se muestra el error.
456         log.warning("El número de nodos es mayor al número de columnas.")
457         # Se obtiene el número de nodos que se deben de eliminar.
458         del_node = num_nodes - num_cols_B
459         # Se genera las matrices B para cada nodo.
460         matrix_B_parts = np.array_split(matrix_B_raw, num_nodes-del_node,
461                                         axis=1)
462         msg_debug = f"Esclavos a eliminar: {del_node}"
463         msg_debug += f"\nEsclavos funcionando: {SLAVE_ADDRESS[:-del_node]}"
464         log.debug(msg_debug)
465     else:
466         # Se define el número de nodos a eliminar.
467         del_node = 1 - num_nodes
468         log.debug(f"Esclavos funcionando: {SLAVE_ADDRESS[:-del_node]}")
469         # Se genera las matrices B para cada nodo.
470         matrix_B_parts = np.array_split(matrix_B_raw, num_nodes, axis=1)
471         # Se comprueba si todas las matrices de los esclavos tienen el mismo
472         # tamaño.
473         if not all([matrix_B.shape == matrix_B_parts[1].shape
```

```
                    for matrix_B in matrix_B_parts[1:]]):
            # Se vuelve a generar las matrices B para cada nodo.
            num_columns = matrix_B_raw.shape[1]//num_nodes
            extra_columns = matrix_B_raw.shape[1]%num_nodes
            # Se define el tamaño de las matrices.
            ajuste  = [num_columns + extra_columns] # Tamaño de la primera.
            # Se define el tamaño de las matrices de los esclavos.
            ajuste += [num_columns + extra_columns + num_columns * (i + 1)
                       for i in range(num_nodes - 2)]
            matrix_B_parts = np.hsplit(matrix_B_raw, ajuste)
            # Se procede eliminar todas las matrices que no tengan alguna de
            # las dimensiones igual a cero.
            matrix_B_parts = [matrix_B for matrix_B in matrix_B_parts
                              if matrix_B.shape[1] != 0]
    # Se define la matriz B que se trabaja localmente.
    matrix_B = matrix_B_parts[0]
    # Se obtiene la cantidad de tiempo que se demora en crear los datos.
    data_creation_time = time.time() - start_time
    msg_info=f"Tiempo de creación de datos: {data_creation_time} segundos"
    log.info(msg_info)
    # ------------------------------------------------------------------
    # * Se define el mensaje a mostrar en el log.*
    # Se muestra mensaje informativo, con las matrices creadas.
    msg_info = f"Matriz A ({num_rows_A}x{num_cols_A}):\n{matrix_A}"
    msg_info += f"Matriz B ({num_rows_B}x{num_cols_B}):\n{matrix_B_raw}"
    log.info(msg_info)
    # Se muestra mensaje para pruebas, con la matrices partidas.
    loc_shape = matrix_B.shape
    msg = f"\nMatriz B (local, {loc_shape[0]}x{loc_shape[1]}):\n{matrix_B}"
    sl_sz = (0,0)
    for i, element in enumerate(matrix_B_parts[1:]):
        sl_sz = element.shape # Tamaño de la matriz del esclavo.
        msg +=f"\nMatriz B (esclavo {i}, {sl_sz[0]}x{sl_sz[1]}):\n{element}"
    log.info(msg)
    # ------------------------------------------------------------------
    # * Se convierte las matrices en arrays de bytes. *
    # Se convierte la matriz A en un arreglo de bytes.
    matrix_A_bytes = matrix_A.tobytes()
    # Se convierte las matrices B en arreglos de bytes.
    matrix_B_bytes = [matrix.tobytes() for matrix in matrix_B_parts[1:]]
    # Obtener las dimensionales de las matrices.
    matrix_shape = (num_rows_A, num_cols_A)+sl_sz
    # Convertir las dimensionales de las matrices en un arreglo de bytes.
    matrix_shape_bytes = bytes(bytearray(matrix_shape))

    # Formato de los datos a enviar.
    #  Estado | Registro  | cantidad de datos|  datos | checksum
    #  4 bits |   4 bits  |      1 byte       | n bytes|  1 byte
    # El checksum se calcula con los datos los primeros 2 bytes son encabezado.
    # Se genera el paquete de las dimensionales de las matrices.
    packet_size = generate_data_packets(REGISTER_SIZE, matrix_shape_bytes)
    # Se genera el paquete de la matriz A.
    log.debug(f"Matriz A de bytes:\n{matrix_A_bytes}")
    packet_matrix_A = generate_data_packets(REGISTER_DATA_A, matrix_A_bytes)
    log.debug(f"Paquetes Matriz A:\n{packet_matrix_A}")
    # Se genera el paquete de las matrices B.
    log.debug(f"Matriz B de bytes:\n{matrix_B_bytes}")
    packets_matrix_B = [generate_data_packets(REGISTER_DATA_B, matrix_B)
                        for matrix_B in matrix_B_bytes]
    log.debug(f"Paquetes Matriz B:\n{packets_matrix_B}")

    # ------------------------------------------------------------------
    # * Se define el objeto de la clase pigpio. *
    gpio = pigpio.pi()
    # Se abre el puerto I2C.
    i2c_handlers = [gpio.i2c_open(1, slave)
                    for slave in SLAVE_ADDRESS[:-del_node]]
    # ------------------------------------------------------------------
```

```
    # * Rutina para enviar la información a los esclavos. *
    # Se toma el tiempo global de envío de información.
    time_start_send = time.time()
    # Se inicializa la lista de tiempos de envío de información.
    list_time_send = []
    aux_time = 0.0
    # Se envía la información a cada esclavo.
    for slave, packet_matrix_B in zip(i2c_handlers, packets_matrix_B):
        # Se envía la información al esclavo y se almacena el tiempo.
        aux_time = send_data_packets(slave, packet_size, packet_matrix_A,
                                     packet_matrix_B, gpio, log)
        list_time_send.append(aux_time)
    del aux_time # Se libera el espacio en memoria.
    # Se obtiene el tiempo global total de envío de información.
    time_end_send = time.time() - time_start_send
    # Se muestra mensaje informativo.
    msg_info = f"Tiempo total de envío de información: {time_end_send} segundos"
    log.info(msg_info)
    # ------------------------------------------------------------------
    # * Rutina para generar las distintas partes de matriz C. *
    # Se calcula la porción de matriz C (local).
    matrix_C_local = np.matmul(matrix_A, matrix_B)
    msg_info = f"Matriz C calculada ({matrix_C_local.shape[0]}x"
    msg_info+= f"{matrix_C_local.shape[1]}):\n{matrix_C_local}"
    log.info(msg_info)
    # Se recibe la información de la matriz C calculada por cada esclavo.
    matrix_received = [[] for _ in range(num_nodes-1)]
    list_index = [i for i in range(num_nodes-1)]
    list_index_aux = []
    # Se intenta recibir la información de los esclavos.
    try:
        while not all(matrix_received): # Mientras las listas estén vacías.
            for index in list_index:
                # Se recibe la información de la matriz C calculada por cada
                # esclavo.
                received = receive_data_packets(i2c_handlers[index], gpio, log)
                if(received): # Si se recibió información.
                    # Se convierte en una arreglo de float32.
                    received_n = np.frombuffer(received, dtype=np.float32)
                    # Se agrega la información recibida a la matriz C.
                    matrix_received[index].append(received_n)
                else:
                    # Se agrega el índice a la lista de índices auxiliar.
                    list_index_aux.append(index)
                time.sleep(0.5)
            # Se actualiza la lista de índices.
            list_index = list_index_aux
            list_index_aux = [] # Se vuelve a inicializar la lista auxiliar.
        # ------------------------------------------------------------------
        # * Rutina para concatenar las distintas partes de matriz C. *
        # Se definen las dimensiones del arreglo de numpy
        # Número de filas de matriz A x Número de columnas parte de
        # matriz B.
        matrix_C_shape = (matrix_A.shape[0], sl_sz[1])
        # Se convierte las matrices recibidas en arreglos de numpy.
        matrix_C_received = [np.array(matrix).reshape(matrix_C_shape)
                             for matrix in matrix_received]
        # Se concatenan la matriz calculada localmente y las matrices recibidas.
        matrix_C = np.concatenate((matrix_C_local, *matrix_C_received), axis=1)
        # Se obtiene el tiempo total.
        total_time = time.time() - start_time
        msg = f"La multiplicación de matrices dio como resultado:\n{matrix_C}"
        msg += f"\nTiempo total de ejecución: {total_time} segundos\n"
        log.info(msg)
        # Se crea el mensaje para mostrar en la terminal.
        msg = "{"
        msg += f'''"Tiempo_creacion":"{data_creation_time}",'''
        msg += f'''"Tiempo_envio":"{time_end_send}",'''
```

```
610             msg += f'''"Tiempo_total":"{total_time}"'''
611             msg += "}"
612             # Se muestra el mensaje en la terminal.
613             print(msg)
614         except KeyboardInterrupt: # Se interrumpe la ejecución del programa.
615             # Se muestra mensaje de error.
616             log.error("Se interrumpió la ejecución del programa.")
617         except Exception as e: # Se captura cualquier excepción.
618             # Se muestra mensaje de error.
619             msg_error = f"Se presentó un error: {e}"
620             msg_error += f"\nEn la línea: {traceback.format_exc()}"
621             log.error(msg_error)
622             print(e)
623         # ------------------------------------------------------------------
624         # Se cierra los puertos I2C.
625         for slave in i2c_handlers:
626             gpio.i2c_close(slave)
627         # Se cierra la instancia de la clase pigpio.
628         gpio.stop()
629         # ------------------------------------------------------------------
630 ##############################################################################
631 if __name__ == "__main__":
632     main()
```

Cuadro 36: Código del nodo maestro.

8.2. Código de nodo esclavo

El código de Python del Cuadro 37 se diseñó para configurar a la Raspberry Pi como un esclavo de I2C. Cuando el código se ejecuta, este procede a esperar a que el nodo maestro envíe la información de las dimensiones de las matrices, una vez que el nodo maestro envía la información de las dimensiones de las matrices, el código procede a esperar a recibir la matriz A y su parte correspondiente de la matriz B. Una vez que el nodo esclavo ha recibido toda la información por parte del nodo maestro, el código procede a realizar el cálculo de su parte de la matriz C, para después esperar a que el nodo maestro pida la parte de la matriz C resultante. En la Figura 21 se puede observar un diagrama de flujo del código del nodo esclavo. Cabe resaltar que el código del nodo esclavo debe de ejecutarse antes de correr el código del nodo maestro, además, el programa no finaliza hasta que el usuario ingrese la combinación de teclas *control* + *c* en la terminal donde se ejecutó el código.

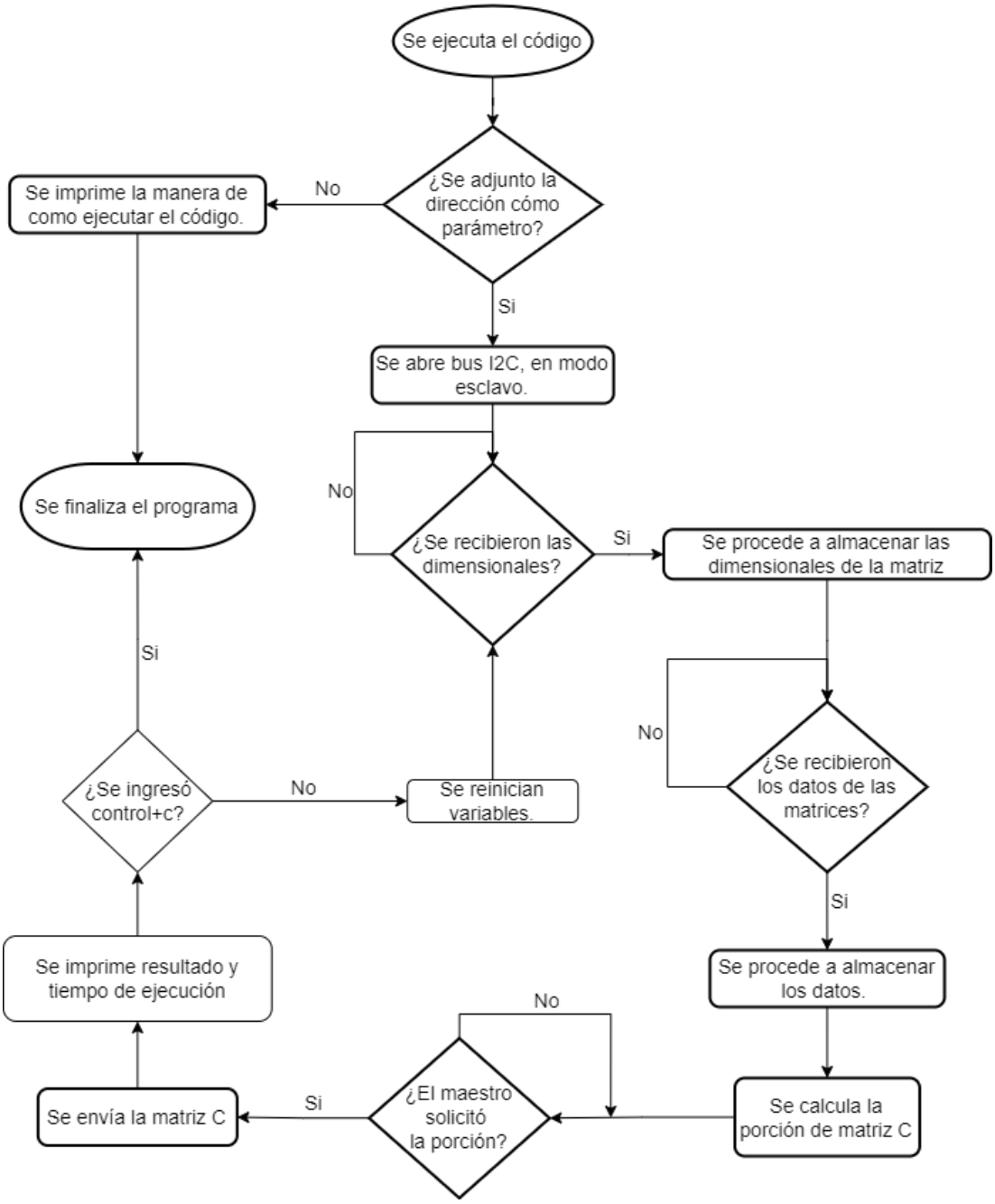


Figura 21: Diagrama de flujo del código de los nodos esclavos.

```
1  ##############################################################################
2  # Autor: Daniel Mundo                                                        #
3  # Fecha: 2024-01-06                                                          #
4  # Descripción:                                                               #
5  #   Programa para el dispositivo esclavo del protocolo i2c. El programa      #
6  #   recibe dos matrices del dispositivo maestro, las multiplica y envía el   #
7  #   resultado al maestro. El programa también registra los tiempos de        #
8  #   ejecución de cada prueba en un archivo de texto.                         #
9  # Uso:                                                                       #
10 #   python3 i2c_slave.py <dirección del esclavo>                             #
11 # Ejemplo:                                                                   #
```

```
#   python3 i2c_slave.py 0x08                                                  #
# Notas:                                                                       #
#   - El programa debe ejecutarse con permisos de administrador.               #
#   - El programa debe ejecutarse en una Raspberry Pi.                         #
#   - El programa debe ejecutarse con Python 3.                                #
#   - El programa debe ejecutarse con la librería pigpio instalada.            #
#   - El programa debe ejecutarse en la Raspberry Pi con la dirección del      #
#     esclavo como argumento.                                                  #
##############################################################################
#                                Librerías                                   #
##############################################################################
import traceback        # Librería para manejo de excepciones.
import pigpio           # Librería para controlar los pines de la Raspberry Pi.
import time             # Librería para gestionar el tiempo.
import sys              # Librería para gestionar los argumentos de entrada.
import numpy as np      # Librería para manejo de matrices.
import logging          # Librería para manejo de logs.
import json             # Librería para manejo de archivos JSON.
##############################################################################
#                                Constantes                                  #
##############################################################################
# Configuración.
IS_ACK_AVAILABLE = True # Bandera para indicar si se debe esperar el ACK.
PACKET_DATA_SIZE = 13   # Tamaño de los paquetes de datos.
PACKET_HEAD_SIZE = 2    # Tamaño de los encabezados de los paquetes.
PACKET_TAIL_SIZE = 1    # Tamaño de las colas de los paquetes.
SDA_PIN          = 2    # Pin de bus de datos.
SCL_PIN          = 3    # Pin de reloj.
# Direcciones de los registros (mismo para maestro y esclavo).
REGISTER_REQUEST = 0x01 # Dirección del registro para solicitar información.
REGISTER_SIZE    = 0x02 # Dirección del registro de tamaño de matrices.
REGISTER_DATA_A  = 0x04 # Dirección del registro de datos de la matriz A.
REGISTER_DATA_B  = 0x08 # Dirección del registro de datos de la matriz B.
# Estados de la comunicación.
START_COMM       = 0xF0 # Inicio de envío de información.
END_COMM         = 0x0F # Finalización de la comunicación.
END_PACKET       = 0x10 # Finalización del paquete de información.
START_OVERFLOW   = 0x20 # Inicio de la comunicación con desbordamiento.
OVERFLOW_DATA    = 0x40 # Bandera para indicar que hay desbordamiento de datos.
##############################################################################
#                                Funciones                                   #
##############################################################################
def checksum(data= bytearray()):
    """
    Calcula el checksum de un arreglo de bytes utilizando un algoritmo
    personalizado.

    Parámetros:
        - data (bytearray): Arreglo de bytes a partir del cual se calculará
                            el checksum.

    Retorna:
        bytes: El checksum calculado como un byte.
    """
    checksum = 0
    for i, byte in enumerate(data):
        checksum ^= byte
        # Se rota el checksum 2 bits a la derecha si el índice es par, caso
        # contrario se rota 2 bits a la izquierda.
        if(i%2 == 0): checksum ^= byte << 2
        else: checksum ^= byte >> 2
    # Se rota el checksum 1 bit a la derecha.
    checksum = (checksum >> 1) & 0xFF
    # Se convierte el checksum a bytes.
    checksum = checksum.to_bytes(1, byteorder='big')
    return checksum # Se retorna el checksum calculado como byte.
def wait_for_ack(value_to_check = bytes(), slave= int(), pi= pigpio.pi(),
                 logs= logging.getLogger(), timeout= 180.0):
```

```
80      """
81      Espera a que se reciba un ACK (acknowledge) del dispositivo esclavo en
82      el bus I2C.
83
84      Parámetros:
85          - slave: Dirección del dispositivo esclavo.
86          - pi: Instancia de la clase pigpio para controlar el bus I2C.
87          - timeout: Tiempo máximo de espera para recibir el ACK.
88          - logs: Instancia de la clase logging para registrar eventos.
89
90      Retorna:
91          - ack: True si se recibió el ACK, False si no se recibió.
92      """
93      # Se define el tiempo inicial.
94      start_time = time.time()
95      # Se define el tiempo actual.
96      current_time = start_time
97      # Se define el estado del ACK.
98      ack = False
99      logs.debug(f"Valor esperado de ACK {value_to_check}")
100     # Se espera a que se reciba el ACK.
101     while (current_time - start_time) < timeout:
102         # Se pregunta si se ha recibido el ACK.
103         (_, n, data) = pi.bsc_i2c(slave)
104         # Si se ha recibido el ACK, se sale del ciclo.
105         if n:
106             logs.debug(f"Toda la data recibida {data}")
107             if value_to_check in data:
108                 logs.debug("Se recibió el ACK")
109                 ack = True
110                 break
111         time.sleep(1)
112         # Se obtiene el tiempo actual.
113         current_time = time.time()
114     if ack == False:
115         # Se muestra mensaje de error si no se recibió el ACK.
116         logs.debug("No se recibió el ACK")
117     return ack # Se retorna el estado del ACK.
118 def generate_data_packets(register=int(), data=bytes()):
119     """
120     Genera paquetes de datos a partir de un registro y una matriz de bytes.
121
122     Parámetros:
123         - register (int): Dirección del registro de donde se obtendrá la
124                           información.
125         - data (bytes): Matriz de bytes a partir de la cual se generarán los
126                         paquetes de datos.
127
128     Retorna:
129         list: Lista de paquetes de datos.
130     """
131     num_bytes=len(data)
132     # Se calcula el número de paquetes de datos a generar.
133     num_packets = num_bytes//PACKET_DATA_SIZE
134     res_packets = num_bytes%PACKET_DATA_SIZE
135     # Se crea una lista vacía para almacenar los paquetes de datos.
136     packets = []
137         # Se genera el paquete del arreglo de bytes.
138     if (abs(num_packets) < 1) or ((num_packets == 1) and (res_packets == 0)):
139         if IS_ACK_AVAILABLE:
140             # Se calcula el checksum.
141             local_checksum = checksum(bytearray(data))
142         else:
143             local_checksum = b''
144         # data_to_send = Tamaño de datos + datos + checksum.
145         data_to_send =  bytes([num_bytes])+data+\
146             local_checksum
147         # Se genera el paquete y se agrega a la lista de paquetes.
```

```
148         packets = [[int(END_PACKET|register),              # Estado y registro.
149                     bytes(data_to_send)]]                  # Campo de data.
150     else:
151         # Se revisa que no existan bytes sobrantes, si hay se agrega otro
152         # paquete más.
153         if res_packets !=0 : num_packets +=1
154         # Se calcula el checksum.
155         if IS_ACK_AVAILABLE:
156             local_checksum = checksum(bytearray(data[:PACKET_DATA_SIZE]))
157         else:
158             local_checksum = b''
159         # Se genera el campo de data.
160         data_to_send = bytes([PACKET_DATA_SIZE])+data[:PACKET_DATA_SIZE]+\
161             local_checksum
162         # Se genera el primer paquete.
163         packets = [[int(START_OVERFLOW|register),           # Estado y registro.
164                     bytes(data_to_send)]]                  # Campo de data.
165         # Se genera el/los paquete(s) de la matriz A.
166         for i in range(0, num_packets-2):
167             # Calculando el checksum.
168             if IS_ACK_AVAILABLE:
169                 # Variable auxiliar para convertir de bytes a bytearray.
170                 aux_b = bytearray(
171                     data[PACKET_DATA_SIZE*(i+1):PACKET_DATA_SIZE*(i+2)]
172                                 )
173                 local_checksum = checksum(aux_b)
174             else:
175                 local_checksum = b''
176             # Se genera el campo de data.
177             data_to_send = bytes([PACKET_DATA_SIZE])+\
178                 data[PACKET_DATA_SIZE*(i+1):PACKET_DATA_SIZE*(i+2)]+\
179                     local_checksum
180 
181             # Se genera el paquete y se agrega a la lista de paquetes.
182             packets.append([int(OVERFLOW_DATA|register),    # Estado y registro.
183                             bytes(data_to_send)            # Campo de data.
184                             ])
185         # Se genera el último paquete.
186         # Variable auxiliar para convertir de bytes a bytearray.
187         aux_b = bytearray(
188                 data[PACKET_DATA_SIZE*(num_packets-1):]
189                                 )
190         # Calculando el checksum.
191         if IS_ACK_AVAILABLE:
192             local_checksum = checksum(aux_b)
193         else:
194             local_checksum = b''
195         # Se genera el campo de data.
196         data_to_send = bytes([len(aux_b)])+\
197             data[PACKET_DATA_SIZE*(num_packets-1):]+local_checksum
198         # Se genera el paquete y se agrega a la lista de paquetes.
199         packets.append([int(END_PACKET|register),          # Estado y registro.
200                         bytes(data_to_send)])              # Campo de data.
201     return packets # Se retorna la lista de paquetes de datos.
202 def send_data_packets(slave=int(), packets=list(), pi=pigpio.pi(),
203                       logs=logging.getLogger()):
204     """
205     Se envía los paquetes de datos al maestro vía I2C.
206 
207     Parámetros:
208         - slave (int): Dirección del dispositivo esclavo.
209         - packets (list): Lista de paquetes de datos a enviar.
210         - pi: Instancia de la clase pigpio para controlar el bus I2C.
211         - logs: Instancia de la clase logging para registrar eventos.
212 
213     Retorna:
214         - total_time (float): Tiempo total de envío de datos.
215     """
```

```
    # Se define la lista de paquetes de datos a enviar.
    packets_to_send = []
    # Se revisa si hay desbordamiento de datos.
    if isinstance(packets[0], list):
        # Se agrega el paquete de la matriz C.
        packets_to_send.extend(packets)
    else:
        # Se agrega el paquete de la matriz C.
        packets_to_send.append(packets)
    # Se define el tiempo inicial de envío de datos.
    start_time = time.time()
    # Inicialización de la variable de datos.
    data = b''
    # Se procede a enviar cada paquete de datos.
    for packet in packets_to_send:
        # Se obtiene la cantidad de bytes del paquete.
        num_bytes =len(packet[1])-2
        # Se prepara el paquete de datos para enviarlo.
        data = bytes(packet[0].to_bytes(1, byteorder='big')+packet[1])
        logs.debug(f"Enviando:\n{data}")
        # Se revisa si el registro del paquete es END_PACKET.
        if (packet[0] & 0xf0) == END_PACKET:
            # Se comprueba si el paquete tiene la longitud correcta.
            if num_bytes != PACKET_DATA_SIZE:
                # Si no lo es se recupera el valor de checksum.
                local_checksum = data[-1:]
                # Se le agregan los bits necesarios para ajustar el tamaño.
                data += bytes(PACKET_DATA_SIZE-num_bytes)
            else:
                # En caso sea de la longitud correcta se recupera el valor de
                # checksum.
                local_checksum = data[-1:]
        else:
            # Si no es END_PACKET se recupera el valor de checksum.
            local_checksum = data[-1:]
        # Se envía el paquete de datos.
        pi.bsc_i2c(slave, data)
        # Se espera a que se reciba el ACK.
        if IS_ACK_AVAILABLE:
            wait_for_ack(local_checksum, slave, pi, logs)
        time.sleep(0.2)
    # Se obtiene el tiempo total de envío de datos.
    total_time = time.time() - start_time
    return total_time # Se retorna el tiempo total de envío de datos.
def setup_logger(name='', log_file=''):
    """
    Función para configuración del log que contiene la información de la
    ejecución del programa.

    Parámetros:
        - name: Nombre del archivo de logs.
        - log_file: Nombre del archivo de logs.

    Retorna:
        - logger: Objeto de la clase logging.
    """
    # Se crea el formato de los logs.
    formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
    # Se crea el archivo de logs.
    handler = logging.FileHandler(log_file)
    # Se agrega el formato al archivo de logs.
    handler.setFormatter(formatter)
    # Se crea el objeto de logs.
    logger = logging.getLogger(name)
    # Configurando el nivel de logs.
    logger.setLevel(logging.INFO)
    # Se agrega el formato al objeto de logs.
    logger.addHandler(handler)
```

```
284     return logger # Se retorna el objeto de logs.
285 def write_json(filename='', data=dict()):
286     """ Escribe los datos en un archivo JSON.
287 
288     Argumentos:
289         filename {str} -- Nombre del archivo JSON (default: {''})
290         data {dict} -- Datos a guardar en el archivo (default: {{}})
291     """
292     # Cargar los datos existentes
293     try:
294         with open(filename, 'r') as file:
295             old_data = json.load(file)
296     except FileNotFoundError:
297         old_data = {}
298     # Actualizar los datos con los resultados de la nueva prueba
299     data_to_save = {**old_data, **data}
300     # Guardar los datos actualizados
301     with open(filename, 'w') as file:
302         json.dump(data_to_save, file, indent=4)
303     return
304 def main():
305     """
306     Función principal del programa.
307     """
308     #-------------------------------------------------------------------------------
309     #* Se obtienen los argumentos de entrada. *#
310     # Se obtiene el nombre del archivo python actual.
311     filename = sys.argv[0].split('/')[-1]
312     # Se obtiene la dirección del dispositivo esclavo.
313     if len(sys.argv) == 2:
314         slave = int(sys.argv[1], base=16)
315     else:
316         print(f"Uso: python3 {filename} <dirección del esclavo>")
317         sys.exit(1)
318     # Se inicia el contador de ejecuciones de la tareas.
319     exec_count = 0
320     #-------------------------------------------------------------------------------
321     #* Se configura el log. *#
322     # Se define el nombre del archivo de logs.
323     log_file = f"/clusterfs/docs/logs/i2c/{filename[:-3]}_{hex(slave)}.log"
324     logs = setup_logger('i2c_slave', log_file)
325     filename_comun = f"/clusterfs/docs/logs/i2c/I2C_{sys.argv[1]}_time.json"
326     #-------------------------------------------------------------------------------
327     #* Se revisa si se puede iniciar el programa. *#
328     # Se revisa si la dirección del dispositivo esclavo es válida.
329     if slave < 0x08 or slave > 0x77:
330         msg_error = "Dirección de esclavo inválida, la dirección debe estar "
331         msg_error += "entre 0x03 y 0x77"
332         # Se muestra mensaje de error.
333         logs.error(msg_error)
334         # Se termina el programa.
335         sys.exit(1)
336     # Inicialización de la variable de datos.
337     buffer_dimensions = []
338     buffer_matrix_a = []
339     buffer_matrix_b = []
340     # Se inicializa el gpio.
341     pi = pigpio.pi()
342     # Se inicializa el bus I2C.
343     pi.bsc_i2c(slave)
344     # Se muestra mensaje de inicio.
345     logs.info(f"\nIniciando Esclavo {hex(slave)}")
346     try:
347         while True:
348             # Se recupera la información del bus I2C.
349             (_, n, data) = pi.bsc_i2c(slave)
350             # Se revisa si se recibió el byte de inicio de comunicación.
351             if((n==1) and (data==bytes([START_COMM]))):
```

```
352                 # Se obtiene el tiempo inicial del programa.
353                 start_time = time.time()
354                 # Se inicializa la bandera de almacenamiento de datos.
355                 flag_save_data = True
356                 # Se muestra mensaje de inicio de comunicación.
357                 logs.debug("Inicio de comunicación")
358                 while flag_save_data:
359                     # Se recupera la información del bus I2C.
360                     (_, n, data) = pi.bsc_i2c(slave)
361                     logs.debug(f"{n},{data}")
362                     if ((n==1) and (data == bytes([END_COMM]))):
363                         logs.debug("Fin de comunicación")
364                         # Se levanta la bandera para salir del ciclo.
365                         flag_save_data = False
366                     elif (abs(n)>1):
367                         if IS_ACK_AVAILABLE:
368                             # Se calcula el checksum.
369                             to_checksum = bytearray(data[2:-1])
370                             local_checksum = checksum(to_checksum)
371                             # Se revisa si el checksum es correcto.
372                             if local_checksum != data[-1:]:
373                                 # Se muestra mensaje de error.
374                                 logs.error("Checksum incorrecto")
375                                 # Se envía el NACK.
376                                 pi.bsc_i2c(slave, local_checksum)
377                                 continue
378                             else:
379                                 # Se envía el ACK.
380                                 pi.bsc_i2c(slave, local_checksum)
381                         logs.debug(f"Se recibió {bytes(data)}")
382                         # Se almacena la información recibida.
383                         first_byte_int = int.from_bytes(data[:1],
384                                                         byteorder='big')
385                         # Se revisa si se recibió el tamaño de las matrices.
386                         if first_byte_int&0x0F == REGISTER_SIZE:
387                             # Se almacena el tamaño de las matrices.
388                             msg_debug = "Se recibió el tamaño de las matrices:"
389                             msg_debug += f" {bytes(data[2:])}"
390                             logs.debug(msg_debug)
391                             # Se revisa si se espera el ACK.
392                             if IS_ACK_AVAILABLE:
393                                 buffer_dimensions.append(data[2:-1])
394                             else:
395                                 buffer_dimensions.append(data[2:])
396                             msg_debug= f"Buffer actual:\n{buffer_dimensions}"
397                             logs.debug(msg_debug)
398                         # Se revisa si se recibió la matriz A.
399                         elif first_byte_int&0x0F == REGISTER_DATA_A:
400                             # Se almacena la matriz A.
401                             msg_debug = "Se recibió la matriz A: "
402                             msg_debug += f"{bytes(data[2:])}"
403                             logs.debug(msg_debug)
404                             # Se revisa si se espera el ACK.
405                             if IS_ACK_AVAILABLE:
406                                 buffer_matrix_a.append(data[2:-1])
407                             else:
408                                 buffer_matrix_a.append(data[2:])
409                             msg_debug= f"Buffer actual:\n{buffer_matrix_a}"
410                             logs.debug(msg_debug)
411                         # Se revisa si se recibió la matriz B.
412                         elif first_byte_int&0x0F ==REGISTER_DATA_B:
413                             # Se almacena la matriz B.
414                             msg_debug = "Se recibió la matriz B: "
415                             msg_debug += f"{bytes(data[2:])}"
416                             logs.debug(msg_debug)
417                             # Se revisa si se espera el ACK.
418                             if IS_ACK_AVAILABLE:
419                                 buffer_matrix_b.append(data[2:-1])
```

```
                            else:
                                buffer_matrix_b.append(data[2:])
                            msg_debug= f"Buffer actual:\n{buffer_matrix_b}"
                            logs.debug(msg_debug)
                        # Se revisa si se recibió un registro inválido.
                        else:
                            # Se muestra mensaje de error.
                            msg_error = "Se recibió un registro inválido: "
                            msg_error += f"{bytes(data)}"
                            logs.error(msg_error)
                    time.sleep(0.1)
                # Se muestra el mensaje que indica el número de ejecución.
                exec_count += 1
                logs.info(f"Ejecución número #{exec_count}")
                # Se obtiene el tiempo total de recepción de datos.
                time_recv = time.time() - start_time
                # Se muestra mensaje del tiempo total de recepción de datos.
                msg_info  = "Tiempo de recepción de datos: "
                msg_info += f"{time_recv} segundos"
                logs.info(msg_info)
                # Se muestra mensaje de inicio de procesamiento de datos.
                logs.debug("Inicio de procesamiento de datos")
                # Se convierte la lista de dimensionales en bytes.
                bytes_dimensions = bytes(buffer_dimensions[0])
                msg_debug= f"Bytes dimensiones:\n{buffer_dimensions}"
                logs.debug(msg_debug)
                # Se obtienen las dimensionales de las matrices.
                dimensions = np.frombuffer(bytes_dimensions, dtype=np.uint8)
                dimensions = dimensions.reshape(2,2) # Se redimensiona.
                msg_debug= f"Dimensionales:\n{dimensions}"
                logs.debug(msg_debug)
                # Se convierte la lista de la matriz A en bytes.
                bytes_matrix_a = b''.join(buffer_matrix_a)
                # Se obtiene la matriz A.
                matrix_a = np.frombuffer(bytes_matrix_a, dtype=np.float32)
                # Se redimensiona la matriz A.
                matrix_a = matrix_a.reshape(dimensions[0,0], dimensions[0,1])
                msg_info = f"Matriz A recibida ({matrix_a.shape[0]}x"
                msg_info+= f"{matrix_a.shape[1]}):\n{matrix_a}"
                logs.info(msg_info)
                # Se convierte la lista de la matriz B en bytes.
                bytes_matrix_b = b''.join(buffer_matrix_b)
                # Se obtiene la matriz B.
                matrix_b = np.frombuffer(bytes_matrix_b, dtype=np.float32)
                # Se redimensiona la matriz B.
                matrix_b = matrix_b.reshape(dimensions[1,0], dimensions[1,1])
                msg_info = f"Matriz B recibida ({matrix_b.shape[0]}x"
                msg_info+= f"{matrix_b.shape[1]}):\n{matrix_b}"
                logs.info(msg_info)
                # Se calcula la matriz C.
                matrix_c = np.matmul(matrix_a, matrix_b)
                # Se convierte la matriz C a un arreglo tipo float32.
                matrix_c = matrix_c.astype(np.float32)
                msg_info = f"Matriz C calculada ({matrix_c.shape[0]}x"
                msg_info+= f"{matrix_c.shape[1]}):\n{matrix_c}"
                logs.info(msg_info)
                # Se convierte la matriz C a bytes.
                bytes_matrix_c = matrix_c.tobytes()
                # Se generan los paquetes de datos.
                packets = generate_data_packets(register=REGISTER_REQUEST,
                                                data=bytes_matrix_c)
                logs.debug(f"Paquetes a enviar:\n{packets}")
                time.sleep(1)
                flag_send_data = False
                while not flag_send_data:
                # Se procede a esperar a que el maestro pregunte por los datos.
                    flag_send_data= wait_for_ack(
                        bytes([END_PACKET|REGISTER_REQUEST]),
```

```
                        slave, pi, logs)
                    time.sleep(1)
                # Se envían los paquetes de datos.
                time_send= send_data_packets(slave, packets, pi, logs)
                # Se toma el tiempo final del programa.
                program_time = time.time() - start_time
                # Se muestra mensaje de fin de procesamiento de datos.
                logs.debug("Fin de procesamiento de datos")
                msg_info  = "Tiempo de envío de datos: "
                msg_info += f"{time_send} segundos"
                msg_info += f"\nTiempo total de ejecución: {program_time} "
                msg_info += "segundos\n"
                logs.info(msg_info)
                # Se inicializa el diccionario para almacenar la información.
                dict_info = {}
                # Se almacena la información del tamaño de la matriz A en un
                # diccionario.
                dim = f"{matrix_a.shape[0]}x{matrix_a.shape[1]}"
                dict_info["Dimension_A"] = dim
                # Se almacena la información del tamaño de la matriz B en un
                # diccionario.
                dim = f"{matrix_b.shape[0]}x{matrix_b.shape[1]}"
                dict_info["Dimension_B"] = dim
                # Se almacena el tiempo de recepción de datos en un diccionario.
                dict_info["Tiempo_Recepcion"] = time_recv
                # Se almacena el tiempo de envío de datos en un diccionario.
                dict_info["Tiempo_Envio"] = time_send
                # Se almacena el tiempo total de ejecución en un diccionario.
                dict_info["Tiempo_Total"] = program_time
                # Se almacena la información en el archivo JSON.
                write_json(filename_comun, {f"Prueba#{exec_count}":dict_info})
                # Se reinician las variables de almacenamiento de datos.
                buffer_dimensions = []
                buffer_matrix_a = []
                buffer_matrix_b = []
            time.sleep(0.05)
    except KeyboardInterrupt:
        logs.info("Programa terminado por el usuario")
        # Se cierra el bus I2C.
        pi.bsc_i2c(0)
        sys.exit(0)
    except Exception as e:
        msg_error  = f"Error inesperado: {e}"
        msg_error += f"\nEn la línea: {traceback.format_exc()}"
        logs.info(msg_error)
        # Se cierra el bus I2C.
        pi.bsc_i2c(0)
        sys.exit(1)
    #-------------------------------------------------------------------------
################################################################################
if __name__ == "__main__":
    main()
```

Cuadro 37: Código de los nodos esclavos.

CAPÍTULO 9

Configuración de *cluster* UART

En la Figura 22 se puede observar el diagrama con la topología del *cluster* UART que se elaboró. La topología cuenta con un nodo maestro y 2 nodos esclavos. Tanto el nodo maestro como los nodos esclavos utilizan los pines de propósito general (GPIO) 14 y 15 para los buses de transmisión y recepción, respectivamente. Debido a que de manera nativa el protocolo UART no soporta la comunicación entre más de dos dispositivos, se tuvo que implementar un multiplexor 4 a 1 para conectar los buses de transmisión de los esclavos al bus de recepción del maestro. Debido a lo anterior el nodo maestro debe de utilizar los pines 5 y 6, para seleccionar al esclavo del cual desea recibir información. Debido a que el protocolo UART como estado inactivo coloca el bus de transmisión en alto, los dos pines del multiplexor que quedan sin utilizar se conectaron al pin 3.3V de la Raspberry Pi. De esta manera, se evitan errores en caso de que se seleccionen esos pines de manera incorrecta.

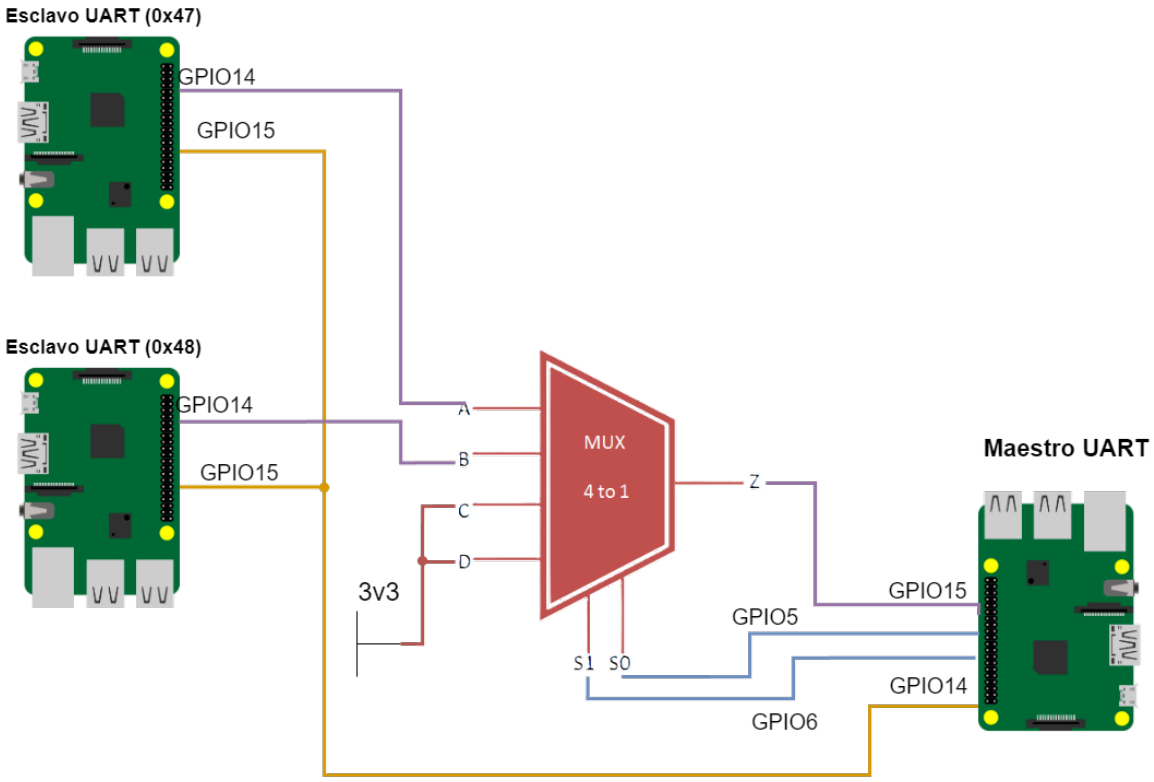


Figura 22: Diagrama del *cluster* UART.

9.1. Código de nodo maestro

El código de Python del Cuadro 38 se diseñó para configurar a la Raspberry Pi como el nodo maestro del *cluster* UART. En la Figura 23 se puede observar el diagrama de flujo del código. Como se observa en el diagrama, lo primero que realiza es la revisión de si las dimensionales de las matrices permiten realizar la multiplicación de matrices y también de que si la cantidad de bytes totales de alguna de las matrices no supere los 255 bytes, debido a que el código genera una serie de paquetes los cuales en el encabezado dedican unicamente 1 byte para indicar la cantidad de bytes de datos que contiene el paquete, por lo que si se superan los 255 bytes de datos, el código no funcionará de manera correcta.

Una vez se verifica lo anterior el código procede a generar las matrices A y B, para después separar la matriz B en partes con igual cantidad de columnas para cada nodo esclavo, en caso el tamaño de matrices no sea divisible entre la cantidad de nodos totales (maestro más esclavos) el nodo maestro se queda con todas las columnas sobrantes. Otro aspecto que toma en cuenta al momento de separar la matriz B, es cuando la cantidad de nodos totales es mayor a la cantidad de columnas de la matriz B, en este caso se eliminan los nodos esclavos sobrantes. Posterior a la separación de la matriz B, el código se encarga convertir las matrices en arreglos de bytes.

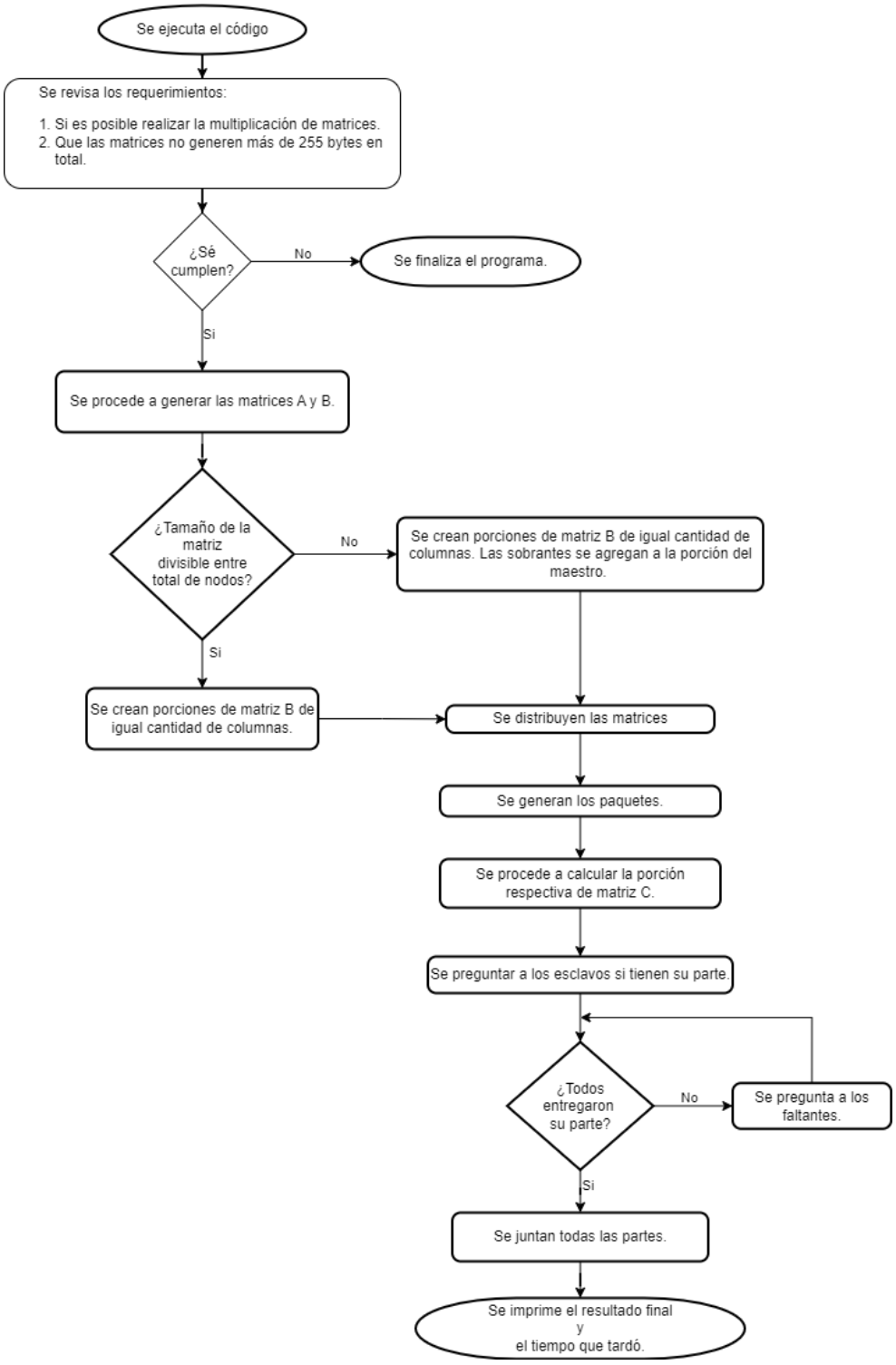


Figura 23: Diagrama de flujo del código del nodo maestro.

Como se mencionó anteriormente, el código genera paquetes para enviar la información de las matrices. Estos paquetes contienen 3 bytes de encabezado la cantidad total de bytes de información. El encabezado consta de 1 byte con la dirección del esclavo, 1 byte con

el tipo de información que se está enviando (dimensionales de las matrices, matriz A o parte de la matriz B) y 1 byte con la cantidad de bytes de información que contiene el paquete. Una vez se generan los paquetes el código se encarga de comenzar la comunicación enviando la información de las dimensiones, seguido de la matriz A y por último la parte correspondiente de la matriz B a cada nodo esclavo. Una vez que los nodos esclavos han recibido la información, el código procede a realizar el cálculo de su parte de la matriz C, para después recolectar las demás partes de la matriz C enviadas por los nodos esclavos y unirlas en una sola matriz C. Finalmente, el código se encarga de imprimir la matriz C resultante y el tiempo total de ejecución del código.

```
1  ##############################################################################
2  # Autor: Daniel Mundo                                                        #
3  # Fecha: 2024-01-06                                                          #
4  # Descripción:                                                               #
5  # Este programa se encarga de enviar la información de las matrices A y B a  #
6  # los esclavos, para que estos realicen la multiplicación de matrices. Este  #
7  # recibe como argumentos de entrada las dimensiones de las matrices A y B,   #
8  # además de la lista de direcciones de los esclavos.                         #
9  # Uso:                                                                       #
10 # python3 uart_master.py kxm mxn <lista de esclavos>                        #
11 # Ejemplo:                                                                   #
12 # python3 uart_master.py 3x2 2x4 0x47,0x48                                   #
13 # Notas:                                                                     #
14 # - El programa debe ejecutarse con permisos de administrador.               #
15 # - Debe de tenerse instalado python3, pigpio, serial y numpy.               #
16 # - La lista de esclavos es opcional, si no se ingresa, se asume que solo hay #
17 #   un esclavo, el 0x47.                                                     #
18 # - La lista de esclavos se ingresa separada por comas, sin espacios.        #
19 # - Las dimensiones de las matrices se ingresan separadas por una x, sin     #
20 #   espacios.                                                                #
21 ##############################################################################
22 #                                 Librerías                                  #
23 ##############################################################################
24 import serial           # Librería para comunicación serial.
25 import pigpio           # Librería para controlar los pines de la Raspberry Pi.
26 import time             # Librería para gestionar el tiempo.
27 import sys              # Librería para gestionar los argumentos de entrada.
28 import numpy as np      # Librería para manejo de matrices.
29 import logging          # Librería para gestionar los logs.
30 ##############################################################################
31 #                                 Constantes                                 #
32 ##############################################################################
33 # Direcciones de los esclavos (disponibles desde 1-255, 0 se usa para
34 # broadcast).
35 DIRECCION_ESCLAVOS = [0x47] # Lista de direcciones de los esclavos.
36 DIRECCION_BROADCAST = 0x00 # Dirección de broadcast.
37 REGISTRO_DIMENSIONES = 0x10 # Registro de dimensiones.
38 REGISTRO_DATOS = [0x41, # Registro de datos para la matriz A.
39                   0x42]   # Registro de datos para la matriz B.
40 BANDERA_INICIO = 0x5B # Bandera de inicio de envío de datos.
41 BANDERA_FIN = 0x5D # Bandera de fin de envío de datos.
42 PINES_MUX = [5, # GPIO 5 o pin 29 de la Raspberry Pi 3B.
43             6]  # GPIO 6 o pin 31 de la Raspberry Pi 3B.
44 ##############################################################################
45 #                               Funciones                                    #
46 ##############################################################################
47 def setup_logger(name='', log_file=''):
48     """Función para configurar los logs.
49
50     Argumentos:
51         name {str} -- Es el nombre del objeto de logs. (default: {''})
52
53         log_file {str} -- Es la ruta del archivo donde se alamacenarán los
54         logs. (default: {''})
55
```

```
56      Retorna:
57          {logger} -- Se retorna el objeto de logs.
58      """
59      # Se crea el formato de los logs.
60      formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
61      # Se crea el archivo de logs.
62      handler = logging.FileHandler(log_file)
63      # Se agrega el formato al archivo de logs.
64      handler.setFormatter(formatter)
65      # Se crea el objeto de logs.
66      logger = logging.getLogger(name)
67      # Configurando el nivel de logs.
68      logger.setLevel(logging.INFO)
69      # Se agrega el formato al objeto de logs.
70      logger.addHandler(handler)
71      return logger # Se retorna el objeto de logs.
72  def generar_paquetes(dirección=int(), registro=int(), datos=bytes(),
73                       log=logging.getLogger()):
74      """
75      Función para generar los paquetes de datos.
76
77      Variables:
78          - dirección: Dirección del esclavo.
79          - registro: Registro de datos.
80          - datos: Datos a enviar.
81          - log: Objeto de logs.
82
83      Retorna:
84          - paquetes: Lista que contiene los bytes del encabezado y los datos.
85      """
86      # Se inicializa la lista de paquetes.
87      paquetes = []
88      # Se calcula la cantidad de bytes del paquete.
89      cantidad_bytes = len(datos)
90      # Se procede a generar el encabezado del paquete.
91      encabezado = bytes([dirección, registro, cantidad_bytes])
92      log.debug(f"Encabezado:\n{encabezado}")
93      # Se procede a agregar el encabezado a la lista de paquetes.
94      paquetes.append(encabezado)
95      # Se procede a agregar los datos a la lista de paquetes.
96      paquetes.append(datos)
97      # Se muestra el paquete generado.
98      log.debug(f"Paquete generado:\n{encabezado+datos}")
99      return paquetes # Se retorna la lista con encabezado y datos.
100
101 def main():
102     """
103     Función principal.
104     Variables:
105         - ser: Objeto de comunicación serial.
106         - matriz: Matriz de punto flotante.
107         - data: Datos recibidos por el puerto serial.
108         - data_float: Datos recibidos por el puerto serial, en punto flotante.
109         - logger: Objeto de logs.
110     """
111     # Se obtiene el tiempo de inicio.
112     tiempo_inicio = time.time()
113     # Se inicializa la variable que almacena los errores.
114     msg_error = ''
115     # TODO: Se configura los pines de la Raspberry Pi 3B.
116     # Se configuran 2 pines de las Raspberry Pi 3B para que funcionen como
117     # como salidas digitales, las cuales se conectan a un MUX 4 a 1.
118     Pi = pigpio.pi()
119     # Se configura los pines como salidas digitales y colocándoles en 0.
120     for pin in PINES_MUX:
121         Pi.set_mode(pin, pigpio.OUTPUT)
122         Pi.write(pin, 0)
123     # Se crea un objeto de tipo serial, en la Raspberry Pi 3B.
```

```
124     # Se establece el baudrate a 115200, se establece el puerto a /dev/ttyS0.
125     # Se establece el tiempo de espera por byte a 10 segundo.
126     ser = serial.Serial(baudrate=115200, port='/dev/ttyS0', timeout=10)
127     #-------------------------------------------------------------------------
128     try:
129         # Se obtienen las dimensiones de la matriz A.
130         dimensiones = sys.argv[1].split('x')
131         # Se define número de filas de la matriz A.
132         filas_A  = int(dimensiones[0])
133         # Se define número de columnas de la matriz A.
134         columnas_A  = int(dimensiones[1])
135         # Se obtienen las dimensiones de la matriz B.
136         dimensiones = sys.argv[2].split('x')
137         # Se define número de filas de la matriz B.
138         filas_B = int(dimensiones[0])
139         # Se define número de columnas de la matriz B.
140         columnas_B = int(dimensiones[1])
141     # Se captura la excepción si no se ingresaron los argumentos de entrada.
142     except IndexError:
143         # Se muestra mensaje de error.
144         msg_error += "No se ingresaron los argumentos de entrada de manera "
145         msg_error += "correcta.\nLos argumentos de entrada deben ser de la "
146         msg_error += "forma: python3 i2c_master.py <filas_A>x<columnas_A> "
147         msg_error += "<filas_B>x<columnas_B>\n"
148         print(msg_error)
149         # Se termina la ejecución del programa.
150         sys.exit(1)
151     # Se revisa si existe un tercer argumento de entrada.
152     if len(sys.argv) > 3:
153         # Se recupera el tercer argumento de entrada, este es la lista de
154         # direcciones de los esclavos.
155         direcciones = sys.argv[3].split(',')
156         # Se reemplaza la lista de direcciones de los esclavos actual, por
157         # la nueva y se convierte en una variable global.
158         global DIRECCION_ESCLAVOS
159         DIRECCION_ESCLAVOS = [int(direccion, 16) for direccion in direcciones]
160 
161     #-------------------------------------------------------------------------
162     # Se obtiene la cantidad de bytes totales de las matrices.
163     limite_bytes_matriz_A = 4 * filas_A * columnas_A
164     limite_bytes_matriz_B = 4 * filas_B * columnas_B
165     # Se comprueba si es posible realizar la multiplicación de matrices.
166     if not (columnas_A == filas_B):
167         # Se muestra un mensaje de error.
168         msg_error += "No es posible realizar la multiplicación de matrices,"
169         msg_error += " debido a que las columnas de la matriz A no es igual"
170         msg_error += " a las filas de la matriz B:"
171         msg_error += f"\n-Matriz A: {sys.argv[1]}\n-Matriz B: {sys.argv[2]}\n"
172     # Se comprueba si la cantidad de bytes totales de las matrices es mayor a
173     # 255 bytes.
174     if (limite_bytes_matriz_A > 255)| (limite_bytes_matriz_B > 255):
175         # Se muestra mensaje de error.
176         msg_error += "La cantidad de bytes totales por matriz es mayor de"
177         msg_error += " 255 bytes."
178         msg_error += f"\nTotal de bytes Matriz A: 4x{sys.argv[1]}="
179         msg_error += f"{4*filas_A*columnas_A}"
180         msg_error += f"\nTotal de bytes Matriz B: 4x{sys.argv[2]}="
181         msg_error += f"{4*filas_B*columnas_B}"
182     #-------------------------------------------------------------------------
183     # * Se configura el Log. *#
184     # Se obtiene el nombre del archivo de python actual.
185     nombre_archivo = sys.argv[0].split('/')[-1]
186     # Se define el nombre del archivo de Logs.
187     ruta_log  = "/clusterfs/docs/logs/uart/"
188     ruta_log += f"{nombre_archivo[:-3]}_MA{sys.argv[1]}_MB{sys.argv[2]}.log"
189     logger = setup_logger('i2c_master', ruta_log)
190     # Se comprueba si existieron errores.
191     if msg_error != '':
```

```
192         # Se muestra mensaje de error.
193         logger.error(msg_error)
194         # Se termina la ejecución del programa.
195         sys.exit(1)
196     else:
197         msg_info  =  "Inicio del calculo de multiplicación de matrices:"
198         msg_info += f"\nMatriz A: {sys.argv[1]}\nMatriz B: {sys.argv[2]}"
199         msg_info += f"\nMatriz C: {filas_A}x{columnas_B}"
200         logger.info(msg_info)
201     #--------------------------------------------------------------------------
202     # Se genera una matriz de punto flotante de tamaño nxm.
203     matriz_A = 100*np.random.rand(filas_A, columnas_A)
204     # Se genera una matriz de punto flotante de tamaño mxk.
205     matriz_B = 100*np.random.rand(filas_B, columnas_B)
206     # Se asegura que la matriz sea de tipo float32.
207     matriz_A = matriz_A.astype(np.float32)
208     # Se asegura que la matriz sea de tipo float32.
209     matriz_B = matriz_B.astype(np.float32)
210     # TODO: Se comprueba el tamaño de la matriz B y la cantidad de nodos.
211     # Se obtiene la cantidad de nodos totales disponibles.
212     cantidad_nodos = len(DIRECCION_ESCLAVOS) + 1
213     if cantidad_nodos > columnas_B:
214         # Si el número de nodos es mayor al número de columnas de la matriz
215         # significa que es necesario eliminar nodos.
216         # Se muestra el error.
217         logger.warning("El número de nodos es mayor al número de columnas.")
218         # Se obtiene el número de nodos que se deben de eliminar.
219         nodos_eliminar = cantidad_nodos - columnas_B
220         # Se genera las matrices B para cada nodo.
221         matrices_B = np.array_split(matriz_B, cantidad_nodos-nodos_eliminar,
222                                     axis=1)
223     else:
224         nodos_eliminar = 1 - cantidad_nodos
225         # Se genera las matrices B para cada nodo.
226         matrices_B = np.array_split(matriz_B, cantidad_nodos, axis=1)
227         # Se comprueba si todas las matrices de los esclavos tienen el mismo
228         # tamaño.
229         if not all([matrix_B.shape == matrices_B[1].shape
230                     for matrix_B in matrices_B[1:]]):
231             # Se vuelve a generar las matrices B para cada nodo.
232             cantidad_columnas = matriz_B.shape[1]//cantidad_nodos
233             extra_columnas = matriz_B.shape[1]%cantidad_nodos
234             # Se define el tamaño de las matrices.
235             # Tamaño de la primera.
236             ajuste  = [cantidad_columnas + extra_columnas]
237             # Se define el tamaño de las matrices de los esclavos.
238             ajuste += [cantidad_columnas + extra_columnas +\
239                 cantidad_columnas * (i + 1) for i in range(cantidad_nodos - 2)]
240             matrices_B = np.hsplit(matriz_B, ajuste)
241             # Se procede eliminar todas las matrices que no tengan alguna de
242             # las dimensiones igual a cero.
243             matrices_B = [matrix_B for matrix_B in matrices_B
244                                    if matrix_B.shape[1] != 0]
245
246     matriz_B_local = matrices_B[0] # Se obtiene la matriz que no se envía.
247     # Se muestra la matriz.
248     logger.debug(f"Matriz B local:\n{matriz_B_local}")
249     # Se muestra el tiempo que demora en generar las matrices.
250     tiempo_creacion_matrices = time.time() - tiempo_inicio
251     # Se muestra el tiempo que demora en generar las matrices.
252     msg_info  = "Tiempo de creación de las matrices: "
253     msg_info += f"{tiempo_creacion_matrices} segundos"
254     # Se muestran las matrices generadas.
255     msg_info += f"\nMatriz A ({sys.argv[1]})\n{matriz_A}"
256     msg_info += f"\nMatriz B ({sys.argv[2]})\n{matriz_B}"
257     logger.info(msg_info)
258     #--------------------------------------------------------------------------
259     # Se convierten las matrices en arreglos de bytes.
```

```
    matriz_A_bytes = matriz_A.tobytes()
    # Se imprime la matriz.
    logger.debug(f"Información a enviar:\n{matriz_A_bytes}")
    matrices_B_bytes = []
    for i, matriz in enumerate(matrices_B[1:]):
        # Se almacenan las matrices convertidas en arreglos de bytes.
        matrices_B_bytes.append(matriz.tobytes())
        # Se muestra la matriz almacenada.
        logger.debug(f"Matriz B esclavo {i+1}:\n{matriz}")
    columnas_matrices_B = matrices_B[1].shape[1]
    #-------------------------------------------------------------------------
    # Se generan los paquete de datos a enviar.
    paquetes = []
    # Se genera el paquete de con las dimensiones de las matrices.
    dimensiones_bytes = bytes([filas_A, columnas_A,
                           filas_B, columnas_matrices_B])
    paquetes.extend(generar_paquetes(DIRECCION_BROADCAST, REGISTRO_DIMENSIONES,
                                     dimensiones_bytes, logger))
    # Se genera el paquete de datos de la matriz A.
    paquetes.extend(generar_paquetes(DIRECCION_BROADCAST, REGISTRO_DATOS[0],
                                     matriz_A_bytes, logger))
    # Se genera el paquete de datos de las matrices B.
    for i, matriz_bytes in enumerate(matrices_B_bytes):
        paquetes.extend(generar_paquetes(DIRECCION_ESCLAVOS[i],
                                         REGISTRO_DATOS[1],
                                         matriz_bytes, logger))
    #-------------------------------------------------------------------------
    # Se obtiene el tiempo inicial de envío de la información.
    tiempo_inicio_envio = time.time()
    # Se envía la información por el puerto serial.
    ser.writelines(paquetes)
    logger.debug(f"Paquete enviado:\n{paquetes}")
    # Se obtiene el tiempo total de envío de la información.
    tiempo_envio = time.time() - tiempo_inicio_envio
    logger.info(f"Tiempo de envío de la información: {tiempo_envio} segundos")
    #-------------------------------------------------------------------------
    # Se establece un tiempo de espera de 1 segundo.
    time.sleep(1)
    #TODO: Se calcula la matriz C local.
    # Se realiza la multiplicación de matrices.
    matriz_C_local = np.matmul(matriz_A, matriz_B_local)
    # Se muestra la matriz.
    logger.info(f"Matriz C local:\n{matriz_C_local}")
    lista_aux = list(zip(matrices_B_bytes, DIRECCION_ESCLAVOS))
    # Se procede a recibir las matrices C de cada esclavo.
    matrices_C = [np.empty((filas_B,columnas_B), dtype=np.float32)
                  for _ in range(len(lista_aux))]
    matriz_de_comparacion = np.empty((filas_B,columnas_B), dtype=np.float32)
    # Se obtiene el tiempo de inicio de recepción de la matriz C.
    tiempo_inicio_C = time.time()
    while True:
        for i in range(len(lista_aux)):
            # Se selecciona el canal del MUX.
            Pi.write(PINES_MUX[0], i&1)
            Pi.write(PINES_MUX[1], i>>1)
            # Se establece un tiempo de espera de 0.5 segundos.
            time.sleep(0.5)
            # Se le envía al esclavo la bandera de inicio.
            msg = bytes([DIRECCION_ESCLAVOS[i], BANDERA_INICIO])
            ser.write(msg)
            # Se lee el puerto serial.
            data = ser.readline()
            # Se muestra la información recibida.
            logger.debug(f"Data:\n{data}")
            if data:
                # Se obtiene la matriz recibida.
                matriz_bytes = data[3:]
                # Se muestra la información recibida.
```

```
328                 logger.debug(f"Matriz recibida:\n{matriz_bytes}")
329                 # Se convierte la matriz a un arreglo de punto flotante.
330                 matriz = np.frombuffer(matriz_bytes, dtype=np.float32)
331                 # Se convierte el arreglo de punto flotante a una matriz.
332                 matriz = matriz.reshape((filas_A, columnas_matrices_B))
333                 # Se muestra la matriz.
334                 logger.debug(f"Matriz recibida:\n{matriz}")
335                 # Se almacena la matriz recibida.
336                 matrices_C[i]=matriz
337         logger.debug(f"Lista de matrices actual:\n{matrices_C}")
338         if all(not np.array_equal(matriz, matriz_de_comparacion)
339                 for matriz in matrices_C):
340             break
341     # Se obtiene el tiempo total de recepción de las matrices C.
342     tiempo_C = time.time() - tiempo_inicio_C
343     logger.info(
344         f"Tiempo de recepción de las matrices C: {tiempo_C} segundos")
345     # Se genera la matriz C.
346     matriz_C = np.concatenate((matriz_C_local, *matrices_C), axis=1)
347     # Se muestra la matriz final.
348     logger.info(f"Matriz C:\n{matriz_C}")
349     # Se obtiene el tiempo total de ejecución.
350     tiempo_total = time.time() - tiempo_inicio
351     logger.info(f"Tiempo total de ejecución: {tiempo_total} segundos\n")
352     # Se crea el mensaje para mostrar en la terminal
353     msg = "{"
354     msg+= f''''"Tiempo_creacion":"{tiempo_creacion_matrices}",'''
355     msg+= f''''"Tiempo_envio":"{tiempo_envio}",'''
356     msg+= f''''"Tiempo_recepcion":"{tiempo_C}",'''
357     msg+= f''''"Tiempo_total":"{tiempo_total}"'''
358     msg+= "}"
359     # Se muestra el mensaje.
360     print(msg)
361     # Se cierra el objeto de comunicación serial.
362     ser.close()
363 if __name__ == "__main__":
364     main()
```

Cuadro 38: Código del nodo maestro.

9.2. Código de nodo esclavo

El código de Python del Cuadro 39 se diseñó para configurar a la Raspberry Pi como un nodo esclavo del *cluster* UART. En la Figura 24 se puede observar el diagrama de flujo del código de los nodos esclavos. Como se puede apreciar en el diagrama de flujo, al momento de ejecutarse el código este espera hasta que el nodo maestro envíe la información a través del bus de recepción. Cuando el nodo maestro envía la información, el código se encarga de detectar que información recibida a través del bus debe de almacenar, ya sean las dimensiones de las matrices, la matriz A o su parte correspondiente de la matriz B. Una vez que el nodo esclavo ha almacenado toda la información que necesita, el código procede a calcular su parte de la matriz C y espera a que el nodo maestro solicite su contribución a la matriz C resultante. Cabe resaltar que el código del nodo esclavo debe de ejecutarse antes de correr el código del nodo maestro y que el programa no finaliza hasta que el usuario ingrese la combinación de teclas *control* + *c* en la terminal donde se ejecutó el código.

Figura 24: Diagrama de flujo del código de los nodos esclavos.

```
1 ##############################################################################
2 # Autor: Daniel Mundo                                                        #
3 # Fecha: 2024-01-06                                                          #
4 # Descripción: Este programa se encarga de recibir dos matrices, las cuales  #
5 # se multiplican y se envía el resultado de la multiplicación.               #
6 # Uso:                                                                       #
7 #   python3 uart_esclavo.py <dirección del esclavo>                          #
8 # Ejemplo:                                                                   #
9 #   python3 uart_esclavo.py 0x08                                             #
```

```
# Notas:                                                                    #
#    - El programa debe ejecutarse con permisos de administrador.          #
#    - El programa debe ejecutarse en una Raspberry Pi.                    #
#    - El programa debe ejecutarse con Python 3.                           #
#    - El programa debe ejecutarse con la librería serial instalada.       #
#    - El programa debe ejecutarse en la Raspberry Pi con la dirección del #
#      esclavo como argumento.                                             #
################################################################################
#                                 Librerías                                    #
################################################################################
import traceback                # Librería para manejo de excepciones.
import serial                   # Librería para comunicación serial.
import sys                      # Librería para obtener los argumentos.
import time                     # Librería para manejo de tiempo.
import numpy as np              # Librería para la creación de matrices.
import logging                  # Librería para la creación de logs.
import json                     # Librería para la creación de archivos JSON.
################################################################################
#                                Constantes                                    #
################################################################################
DIRECCION_BROADCAST = 0x00      # Direcciones de los esclavos (disponibles
                                # desde 1-255, 0 se usa para broadcast).
REGISTRO_DIMENSIONES = 0x10     # Registro de dimensiones de las matrices.
REGISTRO_DATOS = [0x41,         # Registro de datos de la matriz A.
                  0x42]         # Registro de datos de la matriz B.
BANDERA_INICIO = 0x5B           # Bandera de inicio de transmisión.
BANDERA_FIN = 0x5D              # Bandera de fin de transmisión.
DEBUG = False                   # Variable para activar los logs de debug.
################################################################################
#                               Funciones                                      #
################################################################################
def escribir_json(filename='', data=dict()):
    """ Escribe los datos en un archivo JSON.

    Argumentos:
        filename {str} -- Nombre del archivo JSON (default: {''})
        data {dict} -- Datos a guardar en el archivo (default: {{}})
    """
    # Cargar los datos existentes
    try:
        with open(filename, 'r') as file:
            old_data = json.load(file)
    except FileNotFoundError:
        old_data = {}
    # Actualizar los datos con los resultados de la nueva prueba
    data_to_save = {**old_data, **data}
    # Guardar los datos actualizados
    with open(filename, 'w') as file:
        json.dump(data_to_save, file, indent=4)
    return
def configurar_archivo_log(nombre='', ruta_log_1=''):
    """ Esta función se encarga de configurar los archivos de logs.

    Argumentos:
        nombre {str} -- Contiene el nombre del archivo de logs.
        (default: {''})

        ruta_log_1 {str} -- Contiene la ruta del primer archivo de logs.
        (default: {''})

        ruta_log_2 {str} -- Contiene la ruta del segundo archivo de logs.
        (default: {''})

    Retorna:
        logger {logging} -- Es el objeto de logs con el cual se escribirán los
        detalles de los procesos.

        logger_comun {logging} -- Es el objeto de logs con el cual se
```

```
78          escribirán los tiempos de ejecución.
79      """
80      # Se define el formato de los logs.
81      formato_log = '%(asctime)s|%(name)s (%(levelname)s): %(message)s'
82      # Se crea el objeto de logs con el nombre especificado.
83      logger = logging.getLogger(nombre)
84      # Se configura el nivel de logs.
85      if DEBUG: # Si debug es True, se configura el nivel de logs a DEBUG.
86          logger.setLevel(logging.DEBUG)
87      else: # Si debug es False, se configura el nivel de logs a INFO.
88          logger.setLevel(logging.INFO)
89      # Se configura el formato de los logs.
90      formato = logging.Formatter(formato_log)
91      #* Se configura el primer manipulador de logs. *#
92      # Se configura el primer manipulador de logs.
93      manipulador1 = logging.FileHandler(ruta_log_1)
94      # Se configura el nivel del primer manipulador de logs.
95      manipulador1.setLevel(logging.INFO)
96      # Se agrega el formato al primer manipulador de logs.
97      manipulador1.setFormatter(formato)
98      # Se agrega el primer manipulador de logs al objeto de logs.
99      logger.addHandler(manipulador1)
100     return logger # Se retornan los objetos de logs.
101 def generar_paquetes(dirección=int(), registro=int(), datos=bytes(),
102                      log=logging.getLogger()):
103     """Esta función se encarga de generar los paquetes de información que se
104     enviarán por el puerto serial.
105
106     Argumentos:
107         dirección {int} -- Es la dirección del esclavo. (Por defecto: {int()})
108
109         registro {int} -- Es el registro al que se enviará la información.
110         (Por defecto: {int()})
111
112         datos {bytes} -- Es la información que se utilizará para generar el
113         paquete. (Por defecto: {bytes()})
114
115         log {logging} -- Es el objeto de logs.
116         (Por defecto: {logging.getLogger()})
117
118     Retorna:
119         paquetes {list} -- Es una lista que contiene el encabezado y los datos
120         del paquete.
121     """
122     # Se inicializa la lista de paquetes.
123     paquetes = []
124     # Se calcula la cantidad de bytes del paquete.
125     cantidad_bytes = len(datos)
126     # Se procede a generar el encabezado del paquete.
127     encabezado = bytes([dirección, registro, cantidad_bytes])
128     if DEBUG: log.debug(f"Encabezado:\n{encabezado}")
129     # Se procede a agregar el encabezado a la lista de paquetes.
130     paquetes.append(encabezado)
131     # Se procede a agregar los datos a la lista de paquetes.
132     paquetes.append(datos)
133     # Se muestra el paquete generado.
134     if DEBUG: log.debug(f"Paquete generado:\n{encabezado+datos}")
135     return paquetes # Se retorna la lista con encabezado y datos.
136 def recuperar_informacion(slave=int(), data=bytes()):
137     """Esta función se encarga de recuperar la información dirigida al esclavo,
138     la cual se encuentra en el arreglo de bytes.
139
140     Argumentos:
141         slave {int} -- Contiene la dirección del esclavo.
142         (Por defecto: {int()})
143
144         data {list} -- Contiene la información recibida por el puerto serial.
145         (Por defecto: {bytes()})
```

```
146
147     Retorna:
148         slave_data {list} -- Es una lista que contiene la información dirigida
149         al esclavo, la cual se encuentra en el arreglo de bytes.
150     """
151     # Se inicializan las variables.
152     slave_data = [b''] # Almacena la información dirigida al esclavo.
153     while data: # Se revisa la información hasta que no haya más información.
154         # Se obtiene el header de la información.
155         header = data[0:3] # Header= [address, register, number of bytes]
156         if(len(header)!=3): # Se revisa si header es de tamaño 3.
157             break # Se rompe el ciclo.
158         # Se revisa si la dirección recibida es la propia o la de broadcast.
159         if(header[0] in [slave, DIRECCION_BROADCAST]):
160             # Se almacena el paquete de información recibida.
161             slave_data.append(data[0:header[2]+3])
162             # Se actualiza la información recibida.
163             data = data[header[2]+3:]
164         else:
165             # Se actualiza la información recibida.
166             data = data[header[2]+3:]
167     return slave_data[1:] # Se retorna la información dirigida al esclavo.
168 def main():
169     # Se crea un objeto de tipo serial, en la Raspberry Pi 3B.
170     # Se establece el baudrate a 115200, se establece el puerto a /dev/ttyS0.
171     # Se establece el tiempo de espera por byte a 10 segundo.
172     ser = serial.Serial(baudrate=115200, port='/dev/ttyS0', timeout=1)
173     nombre = sys.argv[0].split('/')[-1] # Se obtiene el nombre del archivo
174     # Se obtiene la dirección del esclavo.
175     if len(sys.argv) == 2:
176         direccion_esclavo = [int(sys.argv[1], base=16)]
177     else:
178         # En caso no se reciba la dirección del esclavo, se muestra el uso del
179         # programa y se termina la ejecución.
180         msg_info = "Uso:\n"
181         msg_info += f"sudo python3 {nombre} <dirección del esclavo en hex>"
182         print(msg_info)
183         sys.exit(1)
184     # Comprobar si la dirección del esclavo es válida.
185     if direccion_esclavo[0] not in range(1, 256):
186         print(f"La dirección del esclavo debe estar entre 1 y 255.")
187         sys.exit(1)
188     # Se inicia contador de número de ejecuciones realizadas.
189     contador = 0
190     # Se crea un objeto de logs.
191     # Se define la ruta donde se encontrara almacenado los logs.
192     archivo1 = f"/clusterfs/docs/logs/uart/{nombre[:-3]}_{sys.argv[1]}.log"
193     archivo2 =  f"/clusterfs/docs/logs/uart/UART_{sys.argv[1]}_time.json"
194     logger = configurar_archivo_log(nombre, archivo1)
195     # Se muestra el inicio del programa.
196     logger.info(f"Esclavo {sys.argv[1]} iniciado.")
197     time.sleep(1) # Se da un delay de 1 segundo.
198     Flag_end = False
199     try:
200         while not Flag_end:
201             # Se lee el puerto serial.
202             list_data = []
203             list_data = ser.readlines()
204             if DEBUG: logger.debug(f"Recibido:\n{list_data}")
205             if len(list_data) == 1:
206                 # Condición en la cual se asegura que la variable data se
207                 # tipo bytes.
208                 data = list_data[0]
209             elif len(list_data) > 1:
210                 # Condición para concatenar todas las filas de información en
211                 # caso que existan.
212                 data = b''
213                 for part_data in list_data:
```

```
214                     data += part_data
215             else:
216                 # En caso no se reciba nada se devuelve data, vacía.
217                 data = b''
218                 if DEBUG: logger.debug(f"No hay data Recibida:\n{data}")
219             # Condición en la cual se asegura que la variable data tenga
220             # información.
221             if data:
222                 # Se toma el tiempo de inicio.
223                 tiempo_inicial = time.time()
224                 # Se inicializan las variables.
225                 filas_A = 0 # Define la cantidad de filas de la matriz A.
226                 columnas_A = 0 # Define la cantidad de columnas de la matriz A.
227                 filas_B = 0 # Define la cantidad de filas de la matriz B.
228                 columnas_B = 0 # Define la cantidad de columnas de la matriz B.
229                 tiempo_recepcion = 0.0 # Tiempo de recepción de las dimensiones.
230                 tiempo_matriz_A = 0.0 # Tiempo de recepción de la matriz A.
231                 tiempo_matriz_B = 0.0 # Tiempo de recepción de la matriz B.
232                 # Se inicializan las matrices A y B
233                 matriz_A = np.zeros((1,1), dtype=np.float32)
234                 matriz_B = np.zeros((1,1), dtype=np.float32)
235                 # Se muestra la información recibida.
236                 if DEBUG: logger.debug(f"Bytes recibidos:\n{data}")
237                 # Se obtiene la información dirigida al esclavo.
238                 list_data = recuperar_informacion(direccion_esclavo[0], data)
239                 logger.debug(f"Lista de data:\n{list_data}")
240                 # Se comprueba que si se recibió la información de la matriz B.
241                 if(len(list_data)!=3):
242                     # Si list_data no tiene 3 elementos, significa que no se
243                     # recibió la información de la matriz B, por lo cual se
244                     # se regresa al inicio del ciclo.
245                     continue
246                 # Se muestra que número de ejecución se está realizando.
247                 contador += 1
248                 logger.info(f"Ejecución número: {contador}")
249                 # Se revisa que cada paquete de información.
250                 for data in list_data:
251                     # Se revisa que registro corresponde.
252                     if(data[1]==REGISTRO_DIMENSIONES):
253                         # Se obtienen las dimensiones de las matrices.
254                         filas_A = int.from_bytes(data[3:4], byteorder='big')
255                         columnas_A = int.from_bytes(data[4:5], byteorder='big')
256                         filas_B = int.from_bytes(data[5:6], byteorder='big')
257                         columnas_B = int.from_bytes(data[6:7], byteorder='big')
258                         # Se obtiene el tiempo de recepción de las
259                         # dimensionales respecto al tiempo inicial.
260                         tiempo_recepcion = time.time() - tiempo_inicial
261                         # Se muestra la información recibida.
262                         msg_info  =  f"Tiempo de recepción:"
263                         msg_info  += f" {tiempo_recepcion} segundos."
264                         logger.info(msg_info)
265                         if DEBUG:
266                             msg_debug =  f"Dimensiones recibidas:\n"
267                             msg_debug += f"Filas A: {filas_A}\n"
268                             msg_debug += f"Columnas A: {columnas_A}\n"
269                             msg_debug += f"Columnas B: {columnas_B}"
270                             msg_debug += f"Filas B: {filas_B}"
271                             logger.debug(msg_debug)
272                     elif(data[1]==REGISTRO_DATOS[0]):
273                         # Se obtiene la matriz recibida.
274                         matriz_bytes = data[3:]
275                         # Se muestra la información recibida.
276                         if DEBUG:
277                             logger.debug(f"Matriz A (bytes):\n{matriz_bytes}")
278                         # Se convierte el arreglo de bytes a un arreglo de
279                         # punto flotante.
280                         matriz = np.frombuffer(matriz_bytes, dtype=np.float32)
281                         # Se convierte el arreglo de punto flotante a una
```

```
                        # matriz de tamaño filas x columnas.
                        matriz_A = matriz.reshape((filas_A, columnas_A))
                        # Tiempo de recepción de la matriz A.
                        tiempo_matriz_A = time.time() - tiempo_inicial
                        msg_info = f"Tiempo de recepción de la matriz A:"
                        msg_info += f" {tiempo_matriz_A-tiempo_recepcion}"
                        msg_info += f"\nMatriz A recibida ({filas_A}"
                        msg_info += f"x{columnas_A}):\n{matriz_A}"
                        # Se muestra la matriz.
                        logger.info(msg_info)
                    elif(data[1]==REGISTRO_DATOS[1]):
                        # Se obtiene la matriz recibida.
                        matriz_bytes = data[3:]
                        # Se muestra la información recibida.
                        if DEBUG:
                            logger.debug(f"Matriz B (bytes):\n{matriz_bytes}")
                        # Se convierte el arreglo de bytes a un arreglo de
                        # punto flotante.
                        matriz = np.frombuffer(matriz_bytes, dtype=np.float32)
                        # Se convierte el arreglo de punto flotante a una
                        # matriz de tamaño filas x columnas.
                        matriz_B = matriz.reshape((filas_B, columnas_B))
                        # Se toma el tiempo de recepción de la matriz B.
                        tiempo_matriz_B = time.time() - tiempo_inicial
                        msg_info = f"Tiempo de recepción de la matriz B:"
                        msg_info += f" {tiempo_matriz_B-tiempo_recepcion}"
                        msg_info += f"\nMatriz B recibida ({filas_B}"
                        msg_info += f"x{columnas_B}):\n{matriz_B}"
                        # Se muestra la matriz.
                        logger.info(msg_info)
                    else:
                        if DEBUG: logger.debug(f"Registro desconocido.")
                # Una vez se recupera la información de las matrices, se
                # procede a realizar la multiplicación de matrices.
                # Se realiza la multiplicación de matrices.
                matriz_C = np.matmul(matriz_A, matriz_B)
                # Se muestra la matriz.
                msg_info = f"\nMatriz C ({filas_A}x{columnas_B}):\n{matriz_C}"
                logger.info(msg_info)
                # Se convierte la matriz a un arreglo de bytes.
                matriz_C_bytes = matriz_C.tobytes()
                # Se muestra la matriz.
                if DEBUG: logger.debug(f"Matriz C (bytes):\n{matriz_C_bytes}")
                # Se obtiene el tiempo inicial para el envío de la matriz C.
                tiempo_envio = time.time() - tiempo_inicial
                tiempo_envio_fin = 0
                # Se espera a que el maestro pida la información.
                while True:
                    # Se lee el puerto serial.
                    data = ser.readline()
                    # Se muestra la información recibida.
                    if DEBUG: logger.debug(f"Data:\n{data}")
                    if data:
                        # Se revisa si la dirección recibida es la propia.
                        if(data[0]==direccion_esclavo[0]):
                            # Se revisa si el maestro manda la bandera de
                            # inicio.
                            if(data[1]==BANDERA_INICIO):
                                # Se envía la matriz por el puerto serial.
                                msg = generar_paquetes(direccion_esclavo[0],
                                                       REGISTRO_DATOS[1],
                                                       matriz_C_bytes,
                                                       logger)
                                ser.writelines(msg)
                                if DEBUG:
                                    msg_debug = f"Encabezado+Matriz C:\n{msg}"
                                    msg_debug += "Matriz C enviada:"
                                    msg_debug += f"\n{matriz_C}"
```

```
                                    logger.debug(msg_debug)
                                break
                    time.sleep(1)
                # Se obtiene el tiempo de envío de la matriz C.
                tiempo_envio_fin = time.time() - tiempo_inicial
                tiempo_envio_total = tiempo_envio_fin - tiempo_envio
                msg_info  =  "Tiempo de envío de la matriz C: "
                msg_info += f"{tiempo_envio_total}"
                logger.info(msg_info)
                # Se calcula el tiempo de ejecución total.
                tiempo_ejecucion = time.time() - tiempo_inicial
                msg_info = f"Tiempo de ejecución: {tiempo_ejecucion}\n"
                logger.info(msg_info)
                # Se inicializa el diccionario de información.
                dict_info = {}
                # Se agrega la información al diccionario.
                dict_info['Dimension_A']= f"{filas_A}x{columnas_A}"
                dict_info['Dimension_B']= f"{filas_B}x{columnas_B}"
                dict_info['tiempo_recepcion'] = tiempo_recepcion
                dict_info['tiempo_matriz_A'] = tiempo_matriz_A
                dict_info['tiempo_matriz_B'] = tiempo_matriz_B
                dict_info['tiempo_envio'] = tiempo_envio_total
                dict_info['tiempo_ejecucion'] = tiempo_ejecucion
                escribir_json(archivo2, dict_info)
    except KeyboardInterrupt:
        logger.info(f"Esclavo {sys.argv[1]} terminado.")
        ser.close()
        sys.exit(0)
    except Exception as e:
        msg_error  = f"Error inesperado: {e}"
        msg_error += f"\nEn la línea: {traceback.format_exc()}"
        logger.info(msg_error)
        ser.close()
        sys.exit(1)
if __name__ == "__main__":
    main()
```

Cuadro 39: Código de los nodos esclavos.

CAPÍTULO 10

Resultados

10.1. Prototipo de *cluster* de Raspberry Pi

En la Figura 25 se puede observar el prototipo del *cluster* de Raspberry Pi que se implementó para realizar las pruebas de rendimiento de los protocolos Ethernet, I2C y UART. El prototipo cuenta con un nodo maestro y dos nodos esclavos, los cuales se comunican mediante los protocolos antes mencionados. La conexión entre los nodos se realizó tal y como se mostró en los distintos diagramas de topología expuestos en capítulos anteriores. Para los tres protocolos se utilizó el mismo nodo maestro, siendo este la Raspberry Pi con la USB roja conectada.

En la sección de anexos se puede encontrar el código de Python utilizado para realizar las pruebas de rendimiento de los protocolos Ethernet, I2C y UART. El código de Python se ejecutó en el nodo maestro. Para las pruebas de rendimiento se utilizaron matrices cuadradas con dimensiones de 2×2 hasta 7×7, con el fin de evaluar el comportamiento de los protocolos con distintas dimensiones de matriz, para cada dimensión de matriz se realizaron 10 pruebas. Además, dicho código también se empleó para realizar las pruebas de rendimiento con matrices con dimensión de 100×100, 500×500 y 1000×1000, tanto para el *cluster* Ethernet como para la Raspberry Pi con y sin hilos.

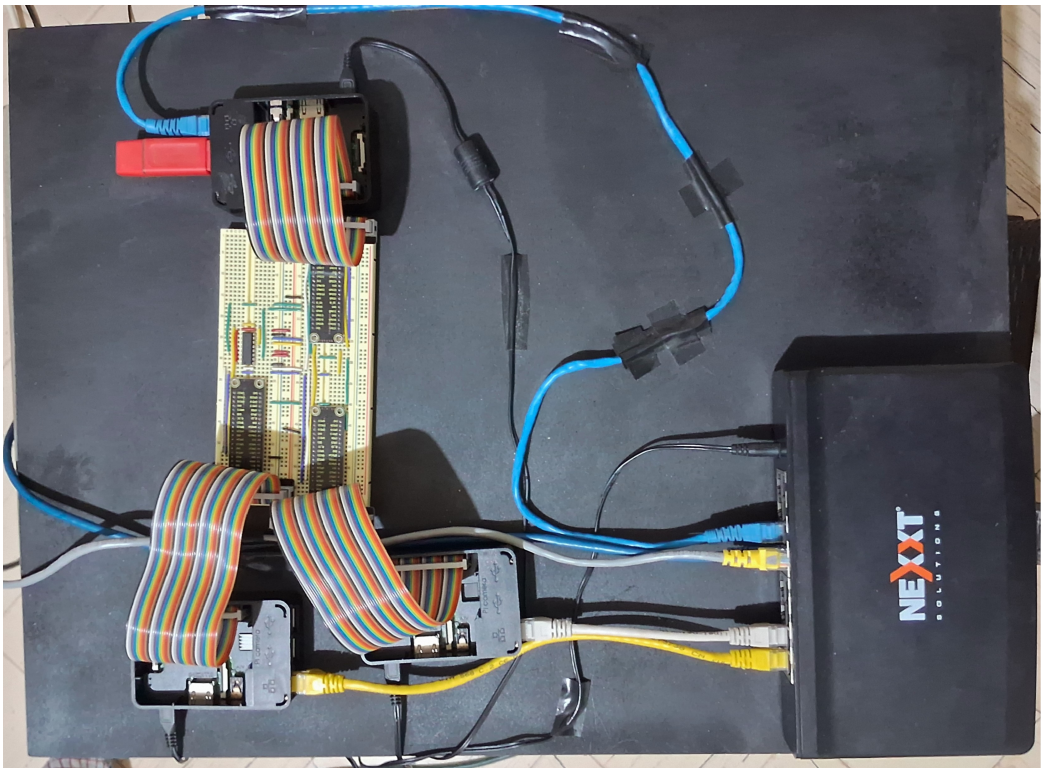


Figura 25: Prototipo del *cluster* de Raspberry Pi.

10.2. Pruebas de rendimiento del protocolo Ethernet

Con los datos obtenidos de las pruebas de rendimiento del protocolo Ethernet, se generó la gráfica de los tiempos de ejecución promedio. En la Figura 26 se puede observar que el tiempo de ejecución tiene un comportamiento cubico que se ajusta a la ecuación $0.0006 \cdot x^3 + 0.0874 \cdot x^2 + 3.968 \cdot x + 2.7545$. Este patrón se respalda con un alto coeficiente de determinación, que alcanza un valor de 0.9065. Además es posible observar como los datos presentan una mayor dispersión a medida que se acercan a la matriz de 5×5 y conforme se alejan la dispersión disminuye. Esto puede ser consecuencia de que al momento de realizar las pruebas en dicha dimensión de matriz el nodo maestro tenga una mayor carga de trabajo y por lo tanto el tiempo de ejecución aumente.

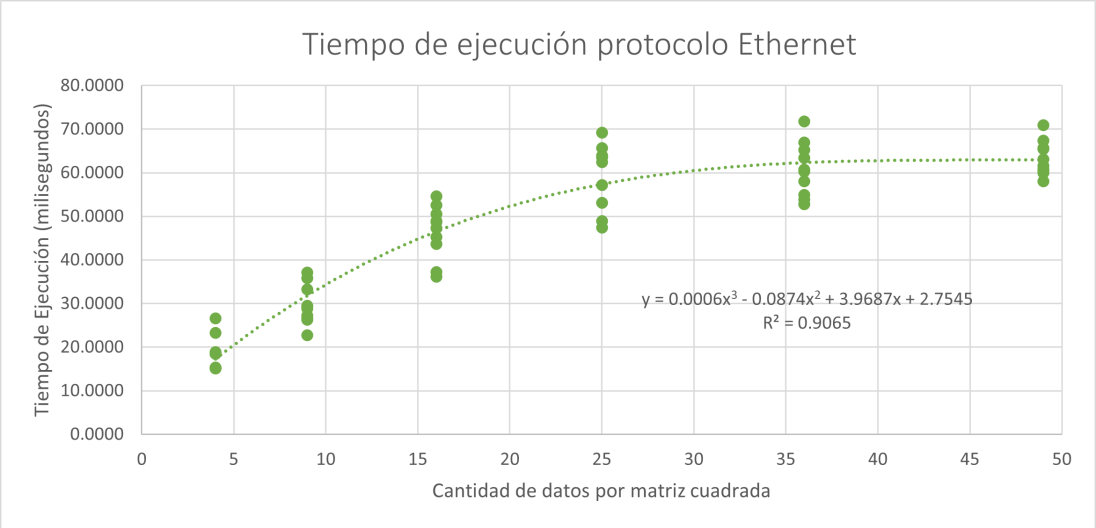


Figura 26: Gráfica de los tiempos de ejecución promedio del protocolo Ethernet.

Pruebas para matrices de dimensiones grandes

A partir de las pruebas de rendimiento con matrices de dimensiones de 100×100, 500×500 y 1000×1000, se originó la gráfica de la Figura 27. En dicha gráfica se puede observar que el tiempo de ejecución tiene un comportamiento lineal que se ajusta a la ecuación $7 \cdot 10^{-6}x + 0.0327$ y que el coeficiente de determinación es de 0.9361. Además es posible observar como los datos presentan una mayor dispersión a medida que se acercan a la matriz de 1000×1000. Esto puede que se deba a que conforme aumenta la dimensión de la matriz, el nodo maestro tenga una mayor carga de trabajo y por lo tanto el tiempo de ejecución incremente.

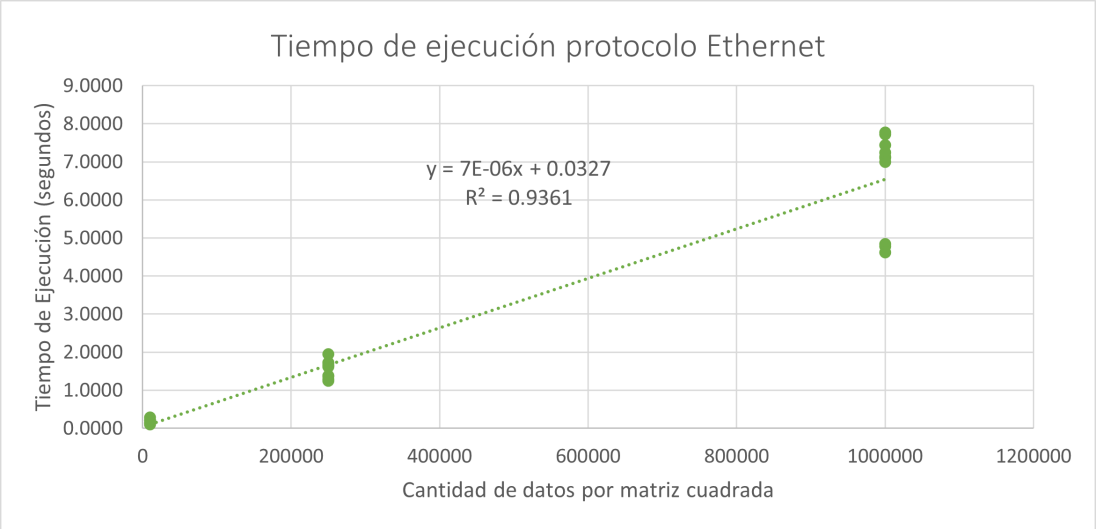


Figura 27: Gráfica de los tiempos de ejecución promedio del protocolo Ethernet.

10.3. Pruebas de rendimiento del protocolo UART

Con la información obtenida a partir de las pruebas de rendimiento del protocolo UART, se generaron dos gráficas del tiempo de ejecución promedio, una para dos nodos ejecutando la operación y otra para tres nodos. En la Figura 28 se puede apreciar que el tiempo de ejecución tiene un comportamiento lineal que se ajusta a la ecuación $0.0008 \cdot x + 23.058$ y que el coeficiente de determinación es de 0.8354. Además, presenta una mayor dispersión en los tiempos a medida que aumenta la dimensión de la matriz. Dicha dispersión puede ser causada por el algoritmo del *cluster* UART, que depende mucho de la sincronización de los tiempos, debido al funcionamiento propio de la librería de Python que se utilizó para la comunicación serial.

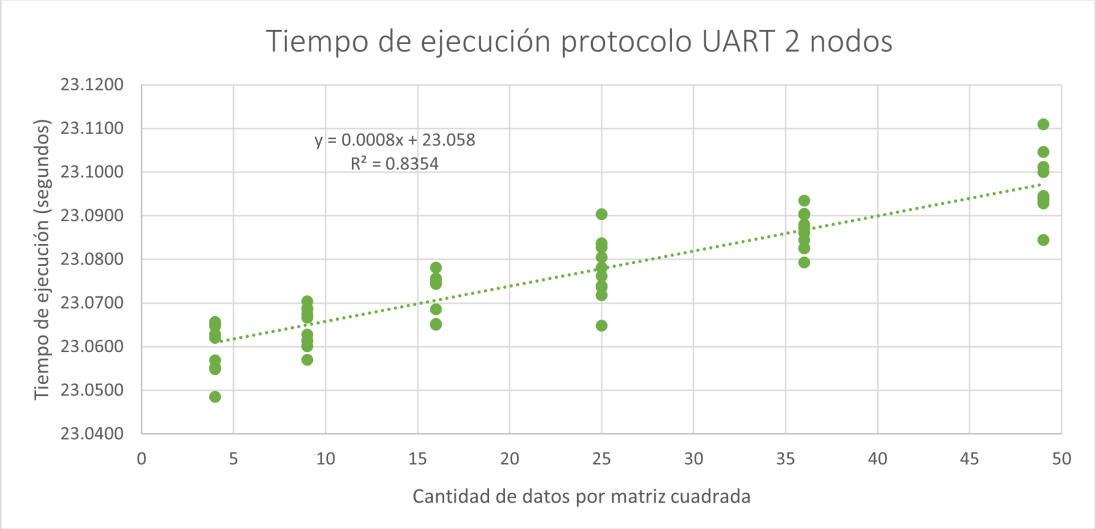


Figura 28: Gráfica de los tiempos de ejecución promedio del protocolo UART con 2 nodos.

En la Figura 29 se puede observar que el tiempo de ejecución no tiene un comportamiento tan predecible, debido a que tiene un coeficiente de determinación de 0.6203 con un modelo lineal. Esto puede deberse a la alta dispersión que tienen los datos, lo cual se puede atribuir a la inclusión del nodo extra con respecto a la gráfica anterior, unido con el hecho de que el protocolo UART no está diseñado para comunicar más de dos dispositivos, esto termina afectando gravemente el rendimiento del protocolo. Además, se debe resaltar que el incluir un nodo extra termina aumentando la dependencia de la sincronización de los tiempos del *cluster* UART.

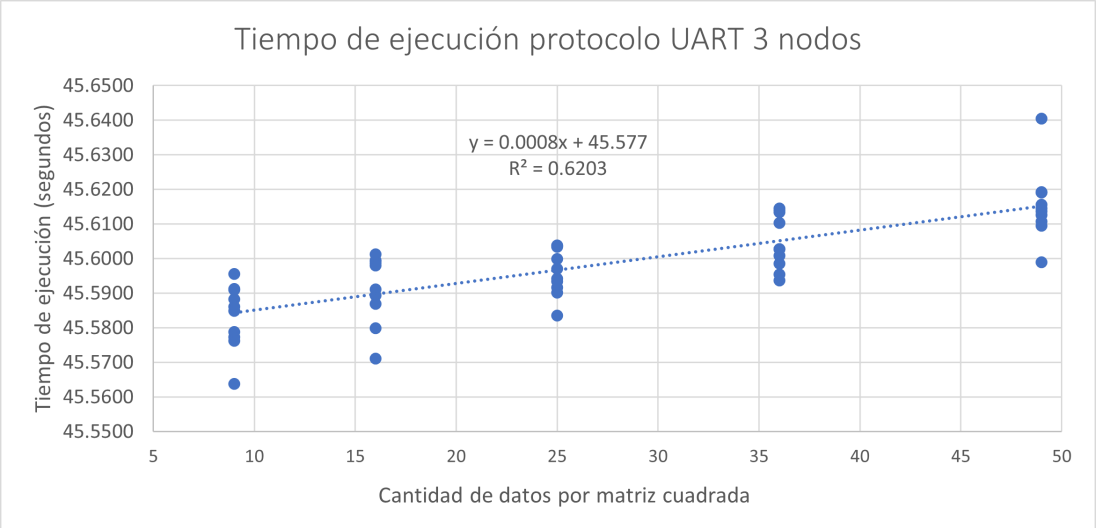


Figura 29: Gráfica de los tiempos de ejecución promedio del protocolo UART con 3 nodos.

10.4. Pruebas de rendimiento del protocolo I2C

Con los datos obtenidos en las pruebas de rendimiento del protocolo I2C, se generaron las gráficas de los tiempos de ejecución promedio de las pruebas de rendimiento del protocolo I2C. En la Figura 30 se puede observar que el tiempo de ejecución para 2 nodos tiene un comportamiento cuadrático que se ajusta a la ecuación $0.0019 \cdot x^2 + 0.8309 \cdot x + 4.1186$. Este patrón se respalda con un alto coeficiente de determinación, que alcanza un valor de 0.9917. Además es posible observar como los datos presentan una menor dispersión a medida que se acercan a la matriz de 5×5 y conforme se alejan la dispersión aumenta. Esto puede deberse a que al momento de realizar las pruebas en esa dimensión de matriz el nodo maestro no presente mayores inconvenientes al momento de enviar las columnas correspondientes.

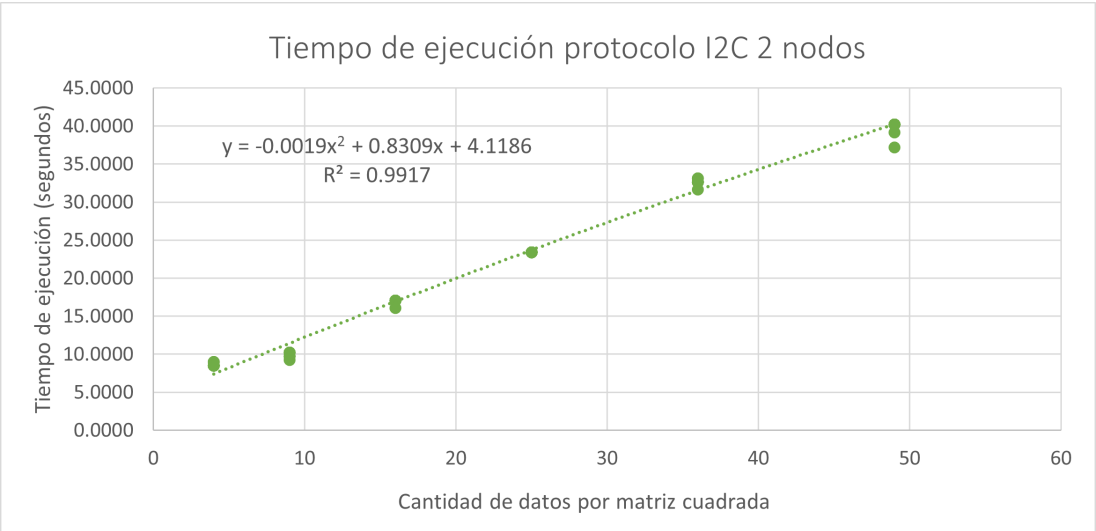


Figura 30: Gráfica de los tiempos de ejecución promedio del protocolo I2C con 2 nodos.

Al observar la Figura 31 se puede apreciar que el tiempo de ejecución para 3 nodos tiene un comportamiento cuadrático que se ajusta a la ecuación $0.0027 \cdot x^2 + 1.1235 \cdot x + 8.5556$. Este patrón se respalda con un alto coeficiente de determinación, que alcanza un valor de 0.9908. Además, es posible apreciar como el modelo es menos preciso que el de 2 nodos, esto puede deberse a que al incluir un nodo extra se aumenta la dependencia de la sincronización del envío de datos, ya que el *cluster* I2C requiere tanto del nodo maestro como de los nodos esclavos para realizar la operación.

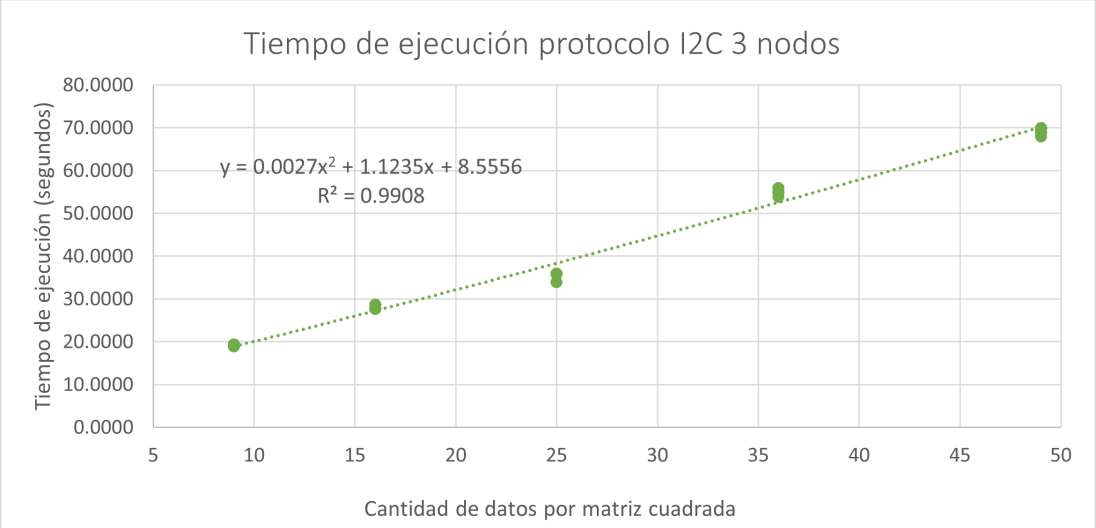


Figura 31: Gráfica de los tiempos de ejecución promedio del protocolo I2C con 3 nodos.

Comparación tiempos actuales y anteriores del protocolo I2C

En el Cuadro 40, se pude apreciar los tiempos de ejecución promedios obtenidos con la primera versión del *cluster* I2C y la versión actual. Se puede observar que los tiempos de ejecución promedio anteriores son significativamente mayores que los tiempos de ejecución promedio actuales, para cada dimensión de matriz. La razón de esta disminución en más del 50 % de los tiempos de ejecución promedio actuales con respecto a los tiempos de ejecución promedio anteriores, se debe a que se implementó un algoritmo más eficiente para la comunicación entre los nodos del *cluster* I2C.

Para el caso de la primera versión del *cluster* I2C se implementó un algoritmo en el cual se enviaba cada dato relacionado con la matriz (elementos y dimensionales) uno por uno, y además se esperaba una confirmación de la recepción del dato antes de enviar el siguiente. Dichos datos se enviaban en paquetes los cuales estaban conformados por un byte con la dirección del esclavo, seguido de un byte que indicaba el registro en donde se iba a almacenar el dato, posteriormente los 4 bytes del dato, y por último un byte de chequeo de errores. Este algoritmo resultó ser ineficiente ya que, se enviaban muchos paquetes de datos lo cual generaba a su vez más tiempos de espera para la confirmación de la recepción de los datos, teniendo como resultado tiempos de ejecución muy altos. Además, otra de las desventajas de este algoritmo es que era necesario enviar un paquete en donde se indicaba el inicio y la finalización de la transmisión de datos, lo cual generaba un tiempo de espera adicional.

Teniendo en cuenta los problemas que presentaba la primera versión, se decidió crear un nuevo algoritmo de comunicación en el cual los paquetes de datos, contenían más de un dato relacionado con la matriz, y además se eliminó la necesidad de enviar un paquete de inicio y finalización de la transmisión de datos. Esto se logró al modificar la estructura de los paquetes de datos, la cual quedó conformada por un byte con la dirección del esclavo, seguido de un byte en el cual sus 4 bits más significativos indicaban el estado de la comunicación (en donde se indica si el paquete es el último o hay más paquetes por recibir) y los 4 bits menos significativos indicaban el registro donde se almacena la información recibida, luego se encuentra un byte que indica la cantidad de bytes a almacenar en el registro, seguido de hasta 10 bytes relacionados con los datos a transmitir y por último un byte de chequeo de errores. Este algoritmo resultó ser más eficiente ya que se enviaban menos paquetes de datos, lo cual generaba menos tiempos de espera para la confirmación de la recepción de los datos, teniendo como resultado tiempos de ejecución más bajos.

Tamaño de la matriz	Tiempo de ejecución promedio anterior (segundos)		Tiempo de ejecución promedio actual (segundos)	
	Nodo maestro	Nodo esclavo	Nodo maestro	Nodo esclavo
2×2	75.25	75.68	8.60	8.98
3×3	107.31	107.76	9.91	10.19
4×4	176.44	175.90	16.95	17.34
5×5	236.55	237.05	23.39	23.54
6×6	347.74	347.29	32.64	32.91
7×7	431.90	431.46	39.78	40.11

Cuadro 40: Tiempos promedio para las pruebas del protocolo I2C.

10.5. Pruebas de rendimiento de la Raspberry Pi con y sin hilos

Con los datos obtenidos de las pruebas de rendimiento de la Raspberry Pi con y sin hilos, se generaron las gráficas de los tiempos de ejecución promedio de las pruebas. En la Figura 32 se puede observar que el tiempo de ejecución, para las Raspberry Pi sin hilos, tiene un comportamiento poco predecible debido a que tiene un coeficiente de determinación de 0.2856 para el polinomio de grado 6, el cual es el coeficiente de determinación más alto obtenido comparándolo con otros modelos. Esto probablemente surge como consecuencia debido a la alta dispersión en los tiempos de ejecución, la cual puede ser resultado de que los tiempos son muy pequeños y que estos sean muy sensibles a cualquier cambio en el sistema.

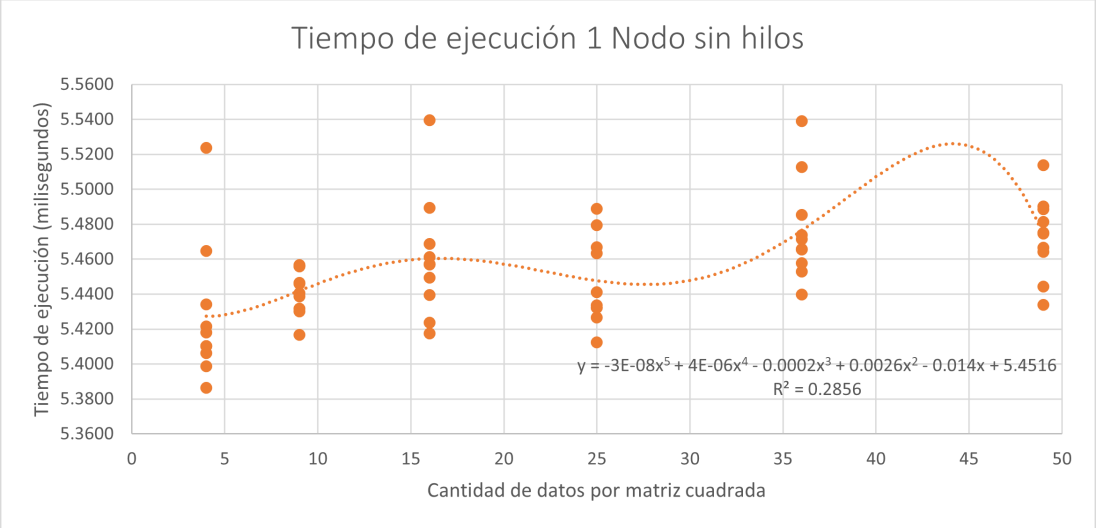


Figura 32: Gráfica de los tiempos de ejecución promedio de la Raspberry Pi sin hilos.

En la Figura 33 se puede observar que el tiempo de ejecución de la Raspberry Pi con 2 hilos tiene un comportamiento igual de impredecible que el de la Figura 32, debido a que tiene un coeficiente de determinación de 0.4075 para el modelo que mejor valor presenta. Esto probablemente también se le pueda atribuir a lo sensible que son los tiempos de ejecución a cualquier cambio en el sistema, debido a lo pequeños que son y con ello provocando que las muestras no sean muy precisas.

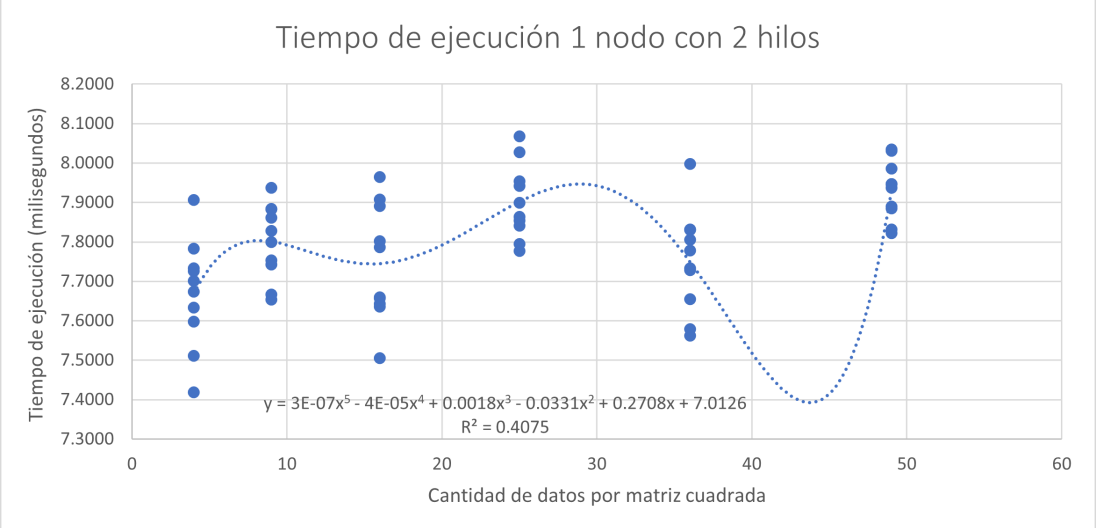


Figura 33: Gráfica de los tiempos de ejecución promedio de la Raspberry Pi con 2 hilos.

Se puede apreciar también un comportamiento similar en la gráfica de la Raspberry Pi con 3 hilos, ver Figura 34, debido a que presenta un coeficiente de determinación de 0.5199 (para el mejor modelo), que a pesar de ser el más alto de los tres, sigue siendo bastante bajo. Cabe resaltar que el caso para 3 hilos presenta los mayores tiempos de ejecución de los tres casos, está subida de tiempo puede estar altamente relacionada al hecho de que al

existir más hilos se genera una mayor carga de trabajo para el procesador, lo cual termina afectando el tiempo de ejecución.

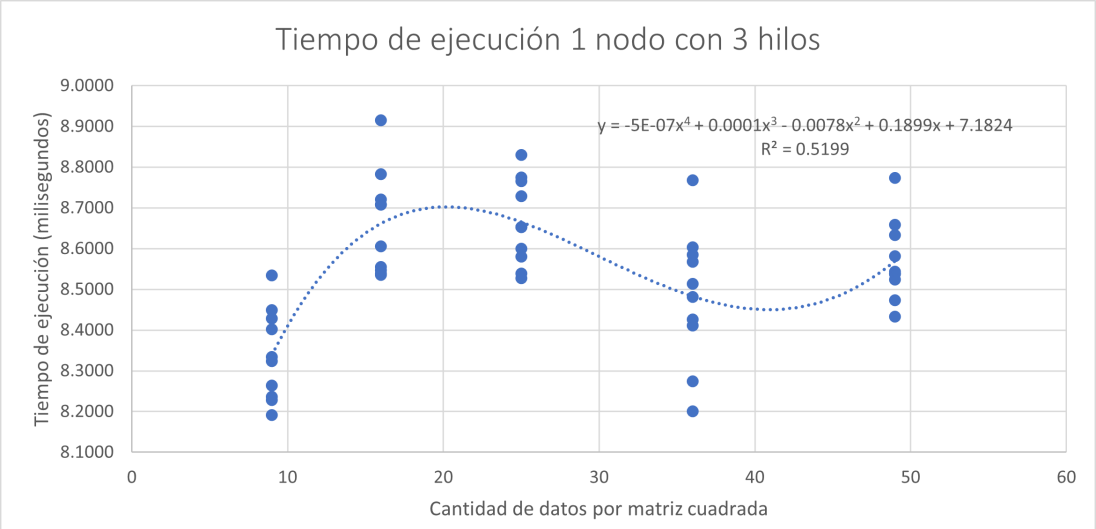


Figura 34: Gráfica de los tiempos de ejecución promedio de la Raspberry Pi con 3 hilos.

Pruebas para matrices de dimensiones grandes

Las pruebas de rendimiento con matrices de dimensiones de 100×100, 500×500 y 1000×1000, se realizaron únicamente para la Raspberry Pi sin hilos y con 2 hilos, ya que al realizar las pruebas con 3 hilos se presentaron problemas. En la Figura 35 se puede observar que el tiempo de ejecución para la Raspberry Pi sin hilos tiene un comportamiento cuadrático el cual se encuentra respaldado por un coeficiente de determinación bastante alto, el cual es visible en dicha gráfica. Además, es posible observar cómo los datos presentan una menor dispersión con respecto a los de la Figura 32, lo cual a su vez refuerza la idea de que los tiempos de ejecución al ser muy pequeños son más sensibles a cualquier cambio en el sistema.

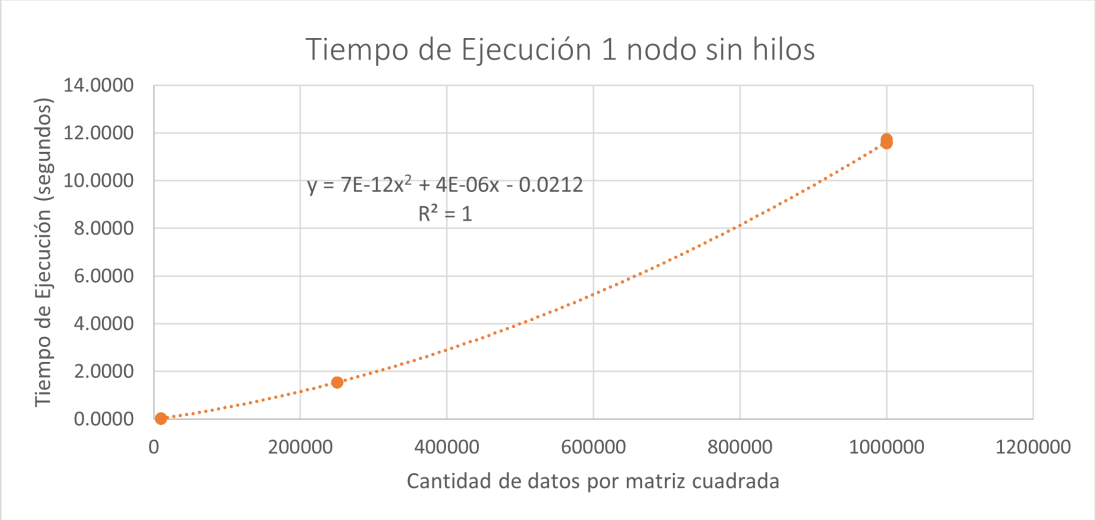


Figura 35: Gráfica de los tiempos de ejecución promedio de la Raspberry Pi sin hilos.

La tendencia de los tiempos antes descrita para la Raspberry Pi sin hilos, se puede observar también en los tiempos de ejecución de la Raspberry Pi con 2 hilos, como se muestra en la Figura 36. En dicha gráfica es posible apreciar como existe una mayor dispersión en los tiempos de ejecución en la matriz de 1000×1000, esto puede deberse a que el procesador de la Raspberry Pi se encuentra sobrecargado al tener que manejar dos hilos de ejecución, lo cual termina afectando el tiempo de ejecución. Además, dicha dispersión termina teniendo un impacto en el coeficiente de determinación, el cual es menor que el de la Raspberry Pi sin hilos.

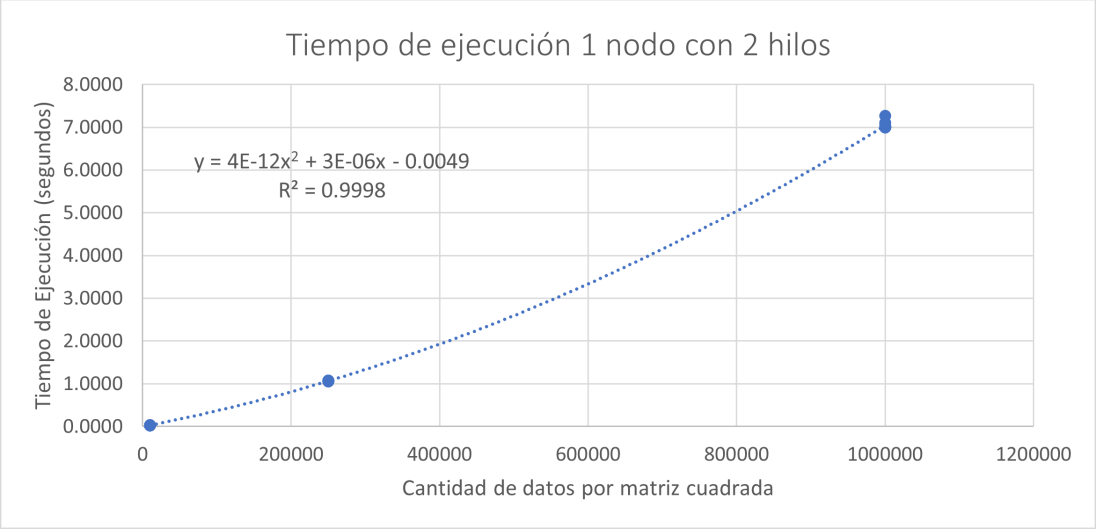


Figura 36: Gráfica de los tiempos de ejecución promedio de la Raspberry Pi con 2 hilos.

10.6. Comparación de las pruebas obtenidas

En la Figura 37, se muestra una comparación de los tiempos de ejecución promedio de los protocolos Ethernet, I2C y UART, junto con los de la Raspberry Pi, tanto con hilos como sin ellos. En la gráfica, se puede observar cómo el protocolo Ethernet presenta los tiempos de ejecución más favorables entre las distintas opciones de comunicación entre los nodos del *cluster* para cada una de las distintas dimensiones. Esto puede atribuirse a que el protocolo Ethernet está diseñado para comunicar dispositivos en diferentes sistemas, lo que requiere una mayor cantidad de datos por paquete para reducir la cantidad de paquetes transmitidos y con ello disminuir el tiempo de transmisión.

Para el protocolo I2C se observa cómo en la gráfica los tiempos van aumentando a medida que incrementa la dimensión de la matriz, esto puede ser causado por el aumento de paquetes de datos que se envían lo cual incrementa los tiempos de espera para la confirmación de la recepción de los datos y con ello aumentado los tiempos de ejecución. En cuanto al protocolo UART, este no parece mostrar cambios significativos en sus tiempos de ejecución promedio a medida que aumenta la dimensión de la matriz. Esto puede atribuirse al hecho de que la cantidad de datos a transmitir no es lo suficientemente grande como para exceder el tiempo de espera del bus serial y con ello afectar el tiempo de ejecución.

En la gráfica también se puede apreciar que la Raspberry Pi sin hilos presenta los menores tiempos de ejecución promedio para todas las dimensiones, seguido de las Raspberry Pi con hilos. Este comportamiento se puede atribuir a que las dimensiones de las matrices no son lo suficientemente grandes, para que el empleo de paralelización de tareas tenga un impacto significativo en el tiempo de ejecución.

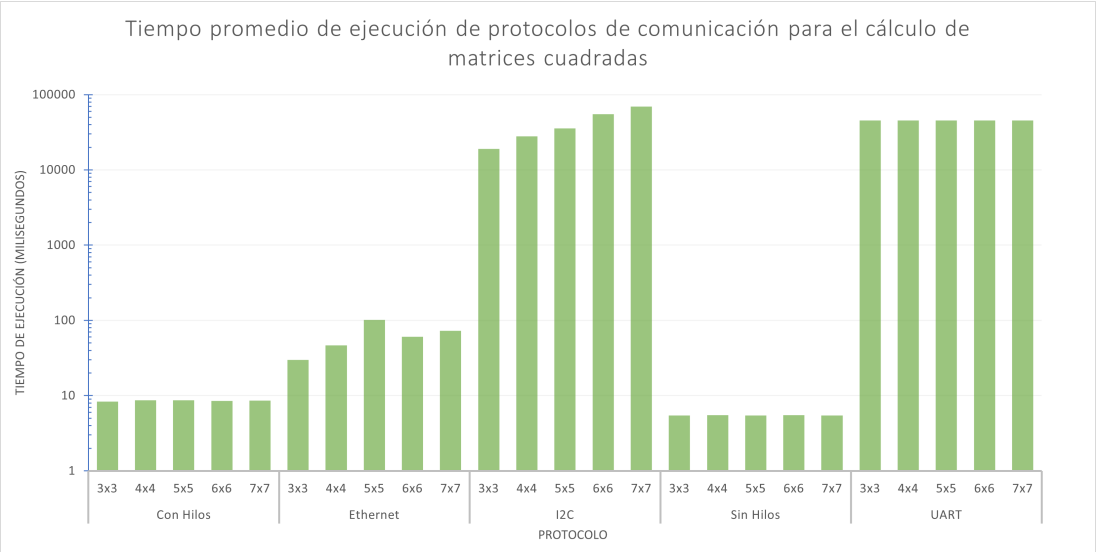


Figura 37: Gráfica de los tiempos de ejecución promedio de los protocolos Ethernet, I2C y UART, además de las de la Raspberry Pi con y sin hilos.

Pruebas para matrices de dimensiones grandes

Con la información obtenida de las pruebas de rendimiento con matrices de dimensiones de 100×100, 500×500 y 1000×1000, se generó la gráfica de la Figura 38. En dicha gráfica se puede apreciar como el protocolo Ethernet presenta los tiempos más favorables conforme aumenta la dimensión de la matriz, seguido de la Raspberry Pi con hilos. Este comportamiento respalda la idea de que el empleo de paralelización de tareas tiene un impacto significativo conforme la dimensión de la matriz aumenta. Para la matriz de 100×100, la Raspberry Pi sin hilos presenta los tiempos de ejecución más favorables, significando esto que la cantidad de datos a procesar no es lo suficientemente grande como para aprovechar el empleo de paralelización de tareas.

En la gráfica, se puede apreciar como la Raspberry Pi sin hilos no presenta los tiempos de ejecución más favorables para la matriz de 500×500. Esto sugiere que la cantidad de datos a procesar es lo suficientemente grande como para justificar el empleo de paralelización de tareas. Para esta dimensión, la versión con hilos presenta los tiempos más favorables. Esto puede deberse a que la cantidad de datos a procesar no es lo suficientemente grande para saturar el procesador de la Raspberry Pi, además, se aprovecha el hecho de que la información no necesita ser enviada a otros nodos para realizar la paralelización de tareas. Y por lo tanto, tener un tiempo de ejecución 50 milisegundos menor que el de los otros dos.

Para la matriz de 1000×1000, el *cluster* Ethernet presenta los tiempos de ejecución más favorables, seguido de la Raspberry Pi con hilos. Este comportamiento respalda la idea de que si la cantidad de datos a procesar es lo suficientemente grande como para saturar el procesador de la Raspberry Pi, el aprovechamiento de la paralelización en el mismo dispositivo no será tan significativo como el envío de datos a otros nodos.

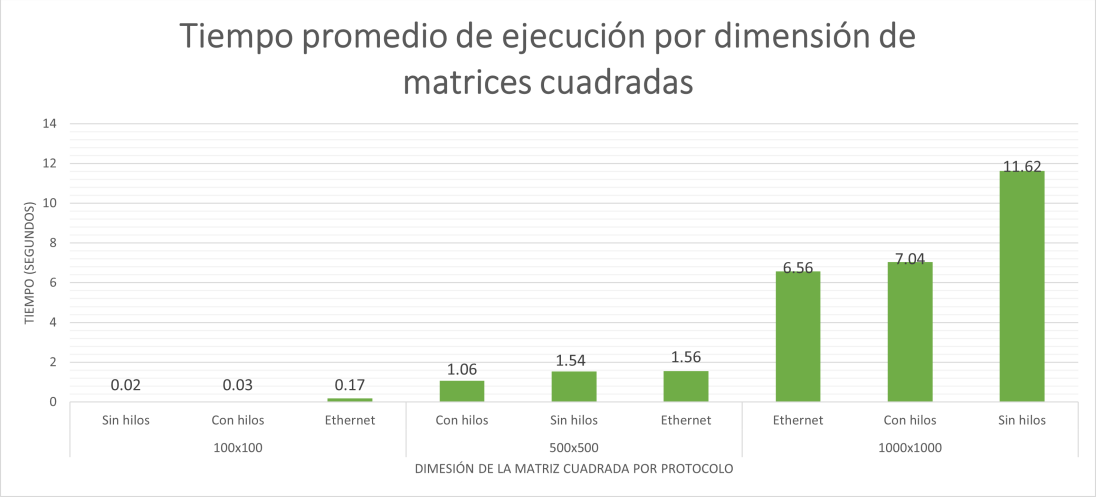


Figura 38: Gráfica de los tiempos de ejecución promedio del *cluster* Ethernet y de la Raspberry Pi con y sin hilos.

CAPÍTULO 11

Conclusiones

1. Se logró con éxito la automatización de la configuración de los nodos esclavos en el *cluster* de Raspberry Pi. Esta automatización incluyó la instalación de paquetes y dependencias necesarias, así como la configuración de parámetros de red y comunicación. Esta mejora facilitará la creación de entornos de procesamiento distribuido, lo que puede tener un impacto positivo en la investigación y desarrollo de proyectos futuros en este campo.

2. La configuración del nodo maestro como servidor DHCP, en su interfaz Ethernet, es un componente clave de la automatización, permitiendo una asignación eficiente de direcciones IP a los nodos esclavos.

3. Se desarrollaron dos códigos, uno para el nodo maestro y otro para el nodo esclavo, que permiten realizar procesamiento distribuido utilizando los protocolo I2C y UART, al ejecutar sus versiones de códigos correspondientes. Estos códigos se probaron con éxito en el *cluster* de Raspberry Pi, demostrando que es posible realizar procesamiento distribuido utilizando los protocolos I2C y UART.

4. Tras analizar los resultados obtenidos en las pruebas de rendimiento, se determinó que el protocolo Ethernet es el más eficiente para la comunicación entre nodos en el *cluster* de Raspberry Pi. Esto se debe a que el protocolo Ethernet tiene un menor tiempo de transmisión de datos en comparación con los protocolos I2C y UART. Además, el protocolo Ethernet tiene una mayor tasa de transferencia de datos en comparación con dichos protocolos.

5. Se logró con éxito la disminución en los tiempos de ejecución del *cluster* I2C, al implementar paquetes de datos en vez de enviar un dato a la vez. Esto permitió disminuir el tiempo de ejecución en un 50 % en comparación con la versión original del código.

6. Se demostró que para matrices de dimensiones pequeñas, la implementación de procesamiento distribuido no es eficiente, ya que el tiempo de comunicación entre nodos es mayor que el tiempo de procesamiento. Sin embargo, para matrices de dimensiones grandes, la implementación de procesamiento distribuido es eficiente, ya que el tiempo de procesamiento es mayor que el tiempo de comunicación entre nodos.

CAPÍTULO 12

Recomendaciones

1. Se recomienda comprobar el rendimiento del *cluster* utilizando una conexión Wifi, como una alternativa cuando no se tiene suficientes puertos Ethernet.
2. Se recomienda investigar otros lenguajes de programación que permitan realizar procesamiento distribuido, como C o C++, para determinar si se obtiene un mejor rendimiento en comparación con Python.
3. Para mejorar el rendimiento del protocolo I2C, se recomienda probar si los tiempos de envío de datos disminuyen si se utiliza un bus I2C con mayor frecuencia de reloj. Esto podría reducir el tiempo de envío de datos y optimizar el uso del bus.
4. Se recomienda investigar si es posible realizar procesamiento distribuido utilizando el protocolo SPI con lenguajes como C o C++.
5. Se recomienda comprobar el rendimiento del *cluster* utilizando otros protocolos que permitan realizar procesamiento distribuido, como el protocolo MPICH.

CAPÍTULO 13

Bibliografía

[1] D. Mencos, "Integración de una computadora central en el Rover UVG para la ejecución de ROS," Tesis de licenciatura, Universidad del Valle de Guatemala, 2022.

[2] C. Bedford, D. Fletchall, B. Jones y D. Kruse, "A Modular Based Design for Distributed Computing Systems," University of Missouri, inf. téc., 2014.

[3] D. Kruse, "A Modular Based Design for Distributed Computing Systems," University of Missouri, inf. téc., 2014.

[4] D. Fletchall, "A Modular Design for Distributed Computing Systems," University of Missouri, inf. téc., 2014.

[5] C. Bedford, "A Minimal Approach to a Monitoring System," University of Missouri, inf. téc., 2014.

[6] B. Jones, "Distributed Computing Using The Raspberry Pi," University of Missouri, inf. téc., 2014.

[7] S. E. Alves Filho, A. M. F. Burlamaqui, R. V. Aroca y L. M. G. Gonçalves, "NPi-Cluster: A Low Power Energy-Proportional Computing Cluster Architecture," *IEEE Access*, vol. 5, págs. 16 297-16 313, 2017. DOI: 10.1109/ACCESS.2017.2728720.

[8] Raspberry Pi, *Raspberry Pi Model B*. dirección: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/ (visitado 10-08-2023).

[9] Raspberry Pi, *Raspberry Pi Documentation - Raspberry Pi hardware*. dirección: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html (visitado 13-03-2023).

[10] *I2C-bus specification and user manual*, NXP Semiconductors, 2021.

[11] S. Afza, *I2C Primer: What is I2C? (Part 1)*, 2016. dirección: https://www.analog.com/en/resources/technical-articles/i2c-primer-what-is-i2c-part-1.html.

[12] M. Harris, *Serial Communications Protocols - Part Two: UART*, Altium, 2021. dirección: https://resources.altium.com/p/serial-communications-protocols-part-two-uart (visitado 23-10-2023).

[13] J. F. Kurose y K. W. Ross, *Computer Networking: A Top-Down Approach*. Pearson, 2020.

[14] R. Garg y G. Verma, *Operating Systems: An Introduction*. Mercury Learning e information, 2017.

[15] J. K. Peckol, *Embedded Systems A Contemporary Design Too*. Wiley, 2019.

[16] "¿Cuál es la diferencia entre modelo OSI y modelo TCP/IP?" (2021), dirección: `https://community.fs.com/es/article/tcpip-vs-osi-whats-the-difference-between-the-two-models.html` (visitado 21-01-2024).

[17] R. Winter, R. Hernandez, G. Chawla et al., "Ethernet Jumbo Frames," Ethernet Alliance, inf. téc., 2009.

[18] S. Siewert y J. Pratt, *Real-Time Embedded Components and Systems with LINUX and RTOS*. Mercury Learning e information, 2016.

[19] B. A. Forouzan, *Data communications and networking*. McGraw-Hill, 2017.

[20] *iptables(8) - Linux man page*, 2023. dirección: `https://linux.die.net/man/8/iptables` (visitado 03-10-2023).

[21] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, jun. de 2021. dirección: `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

[22] SchedMD. "Slurm Workload Manager - Overview." (2021), dirección: `https://slurm.schedmd.com/overview.html` (visitado 30-08-2023).

CAPÍTULO 14

Anexos

14.1. Código para pruebas de rendimiento de la Raspberry Pi

```
##############################################################################
# Autor: Daniel Mundo                                                        #
# Fecha: 2024-01-06                                                          #
# Descripción:                                                               #
# Este programa se encarga de realizar la multiplicación de matrices,        #
# este puede emplear multiprocesamiento o no, dependiendo de los parámetros  #
# de entrada.                                                                #
# Uso:                                                                       #
# python3 rpi_mult_matrices.py nxm mxk -m<cantidad de hilos>                 #
# Ejemplo:                                                                   #
# python3 matrix.py 3x3 3x3 -m4                                              #
# Notas:                                                                     #
# - Se debe tener instalado python3, numpy y threading.                      #
# - Para implementar el multiprocesamiento debe de colocar el parámetro -m   #
#   seguido de la cantidad de hilos a utilizar.                              #
##############################################################################
#                                 Librerías                                  #
##############################################################################
import sys                              # Librería para manejar argumentos.
import numpy as np                      # Librería para manejar matrices.
import threading                        # Librería para manejar hilos.
import time                             # Librería para manejar el tiempo.
import logging                          # Librería para manejar logs.
from pythonjsonlogger import jsonlogger # Librería para manejar logs.
import json                             # Librería para manejar archivos JSON.
##############################################################################
#                                 Funciones                                  #
##############################################################################
def configurar_logs_json(nombre, ruta_del_archivo):
    """ Esta función se encarga de configurar los logs en formato JSON.

    Argumento:
        nombre {str} -- Es el nombre del logger.
        ruta_del_archivo {str} -- Es la ruta del archivo de logs.

    Retorna:
        {logger} -- Es el logger configurado.
    """
    logger = logging.getLogger(nombre)
```

```
40      logger.setLevel(logging.INFO)
41
42      logHandler = logging.FileHandler(ruta_del_archivo)
43      formatter = jsonlogger.JsonFormatter(
44          fmt="%(asctime)s %(levelname)s",
45          json_encoder=json.JSONEncoder,
46          json_default=str,
47          json_indent=4
48      )
49      logHandler.setFormatter(formatter)
50      logger.addHandler(logHandler)
51      return logger
52  ################################################################################
53  #                                 Clases                                       #
54  ################################################################################
55  class esclavos(threading.Thread):
56      """ Esta clase se encarga de realizar la multiplicación de matrices.
57
58      Argumentos:
59          threading {Thread} -- Es la clase de hilos de python.
60      """
61      def __init__(self, id, matriz_A, matriz_B, matriz_C, tiempo=time.time()):
62          """ Esta función inicializa la clase de hilos hijos.
63
64          Argumentos:
65              id {int} -- Es el identificador del hilo.
66              matriz_A {np.array} -- Es la matriz A.
67              matriz_B {np.array} -- Es la matriz B.
68              matriz_C {np.array} -- Es la matriz C.
69          """
70          # Se inicializa la clase de hilos.
71          threading.Thread.__init__(self)
72          # Se inicializan las variables.
73          self.id = id
74          self.matriz_A = matriz_A
75          self.matriz_B = matriz_B
76          self.matriz_C = matriz_C
77          self.tiempo_ejecucion = 0.0
78          self.tiempo_inicio = tiempo
79      def run(self):
80          """ Esta función se encarga de realizar la multiplicación de matrices.
81          """
82          time.sleep(0.005)
83          # Se calcula la matriz resultante.
84          self.matriz_C = np.dot(self.matriz_A, self.matriz_B)
85          # Se obtiene el tiempo de ejecución.
86          self.tiempo_ejecucion = time.time() - self.tiempo_inicio
87          # Se devuelve la matriz resultante.
88          return self.matriz_C
89  ################################################################################
90  if __name__ == "__main__":
91      # Se inicializa el logger.
92      logger = configurar_logs_json("log", "log.json")
93      # Se inicializan las variables.
94      hilos = 1
95      # Se crea el mensaje de uso.
96      msg_error="Uso: python3 rpi_mult_matrices.py nxm mxk -m<cantidad de hilos>"
97      # Se revisa la cantidad de argumentos.
98      if len(sys.argv) != 3 and len(sys.argv) != 4:
99          logger.error(msg_error)
100         print(msg_error)
101         exit(1)
102     # Se revisa si se va a usar multi procesamiento.
103     if len(sys.argv) == 4:
104         # Se revisa si el parámetro -m es correcto.
105         if sys.argv[3][0:2] != "-m":
106             logger.error(msg_error)
107             print(msg_error)
```

```
108             exit(1)
109         # Se revisa si la cantidad de hilos es correcta.
110         try:
111             hilos = int(sys.argv[3][2:])
112         except:
113             msg_error = "La cantidad de hilos debe ser un número entero."
114             logger.error(msg_error)
115             exit(1)
116         if hilos <= 0:
117             msg_error = "La cantidad de hilos debe ser un número entero mayor"
118             msg_error+=" a 0."
119             logger.error(msg_error)
120             exit(1)
121     # Se recuperan las dimensiones de las matrices.
122     try:
123         dim1 = sys.argv[1].split("x")
124         dim2 = sys.argv[2].split("x")
125         dim_matriz_A = [int(x) for x in dim1]
126         dim_matriz_B = [int(x) for x in dim2]
127     except ValueError:
128         msg_error = "Las dimensiones de las matrices deben ser números enteros"
129         logger.error(msg_error)
130         exit(1)
131     # Se revisa si las dimensiones son correctas.
132     if dim_matriz_A[1] != dim_matriz_B[0]:
133         msg_error = "Las dimensiones de las matrices no son correctas."
134         msg_error += "La matriz A debe tener el mismo número de columnas que "
135         msg_error += "el número de filas de la matriz B."
136         logger.error(msg_error)
137         exit(1)
138     # Se inicializa la lista de hilos.
139     lista_hilos = []
140     # Se inicializa la lista de matrices resultantes.
141     matrices_C = []
142     # Se inicializa el identificador de los hilos.
143     id = 1
144     # Se inicializa el tiempo de inicio.
145     tiempo_inicio = time.time()
146     # Se inicializa el tiempo de fin.
147     tiempo_fin = 0
148     # Se inicializa la lista de tiempos de ejecución.
149     tiempos_ejecucion = []
150     # Se inicializa el tiempo de ejecución.
151     tiempo_ejecucion = 0
152     # Se inicializa el tiempo de ejecución total.
153     tiempo_ejecucion_total = 0
154     # Se generan las matrices.
155     matriz_A = 100*np.random.rand(dim_matriz_A[0], dim_matriz_A[1])
156     matriz_B = 100*np.random.rand(dim_matriz_B[0], dim_matriz_B[1])
157     # Se inicializa la matriz C.
158     matriz_C = np.zeros((dim_matriz_A[0], dim_matriz_B[1]))
159     # Se inicializa el diccionario de información.
160     dict_info = {}
161 
162     dict_info["Matriz_A"] = matriz_A.tolist()
163     dict_info["Dimensiones_Matriz_A"] = matriz_A.shape
164     dict_info["Matriz_B"] = matriz_B.tolist()
165     dict_info["Dimensiones_Matriz_B"] = matriz_B.shape
166 
167     if hilos > 1 :
168         # Se revisa si hay multi procesamiento y si la cantidad de hilos es
169         # menor o igual a la cantidad de columnas de la matriz B.
170         if hilos > dim_matriz_B[1]:
171             # Si la cantidad de hilos es mayor a la cantidad de columnas de la
172             # matriz B, se eliminan tantos hilos como sea necesario para que la
173             # cantidad de hilos sea igual a la cantidad de columnas de la
174             # matriz B.
175             hilos = dim_matriz_B[1]
```

```
176             # Muestra un mensaje de advertencia.
177             msg_warning = "La cantidad de hilos es mayor a la cantidad de "
178             msg_warning += f"columnas de la matriz B, se utilizarán {hilos}"
179             msg_warning += "hilos."
180             logger.warning(msg_warning)
181             # Se separa la matriz B.
182             matrices_B = np.array_split(matriz_B, hilos, axis=1)
183         else:
184             # Se separa la matriz B.
185             matrices_B = np.array_split(matriz_B, hilos, axis=1)
186             # Se comprueba que todos los hilos hijos tengan la misma cantidad de
187             # columnas.
188             if not all(matrices_B[0].shape[1] == x.shape[1]
189                     for x in matrices_B):
190                 # Se vuelve a separar la matriz B.
191                 numero_columnas = dim_matriz_B[1]//hilos
192                 columnas_extra = dim_matriz_B[1]%hilos
193                 # Se define el tamaño de las matrices.
194                 dimension = [numero_columnas + columnas_extra]
195                 dimension += [numero_columnas + columnas_extra +\
196                     numero_columnas*(i+1) for i in range(hilos-2)]
197                 # Se separa la matriz B.
198                 matrices_B = np.hsplit(matriz_B, dimension)
199         # Se crean los hilos.
200         for i in range(hilos):
201             dict_info1 = {}
202             dict_info1["Matriz_B"] = matrices_B[i].tolist()
203             dict_info1["Dimensiones_Matriz_B"] = matrices_B[i].shape
204             dict_info["Hilo_"+str(i+1)] = dict_info1
205             # Se crea el hilo.
206             hilo = esclavos(id, matriz_A, matrices_B[i], matriz_C)
207             # Se inicia el hilo.
208             hilo.start()
209             # Se agrega el hilo a la lista de hilos.
210             lista_hilos.append(hilo)
211             # Se aumenta el identificador.
212             id += 1
213         # Se espera a que los hilos terminen.
214         for hilo in lista_hilos:
215             dict_info1 = {}
216             # Se espera a que el hilo termine.
217             hilo.join()
218             # Se agrega la matriz resultante a la lista de matrices
219             # resultantes.
220             matrices_C.append(hilo.matriz_C)
221             dict_info1["Matriz_C"] = hilo.matriz_C.tolist()
222             dict_info1["Dimensiones_Matriz_C"] = hilo.matriz_C.shape
223             # se agrega el tiempo de ejecución a la lista de tiempos de
224             # ejecución.
225             tiempos_ejecucion.append(hilo.tiempo_ejecucion)
226             dict_info1["Tiempo_Ejecucion"] = hilo.tiempo_ejecucion
227             dict_info["Hilo_"+str(hilo.id)] = dict_info1
228 
229         # Se genera la matriz resultante.
230         matriz_C = np.hstack(matrices_C)
231         # Se obtiene el tiempo de ejecución total.
232         tiempo_ejecucion_total = time.time() - tiempo_inicio
233         dict_info["Hilo_"+str(1)]["Tiempo_Ejecucion"] = tiempo_ejecucion_total
234 
235         # Se muestra la matriz resultante.
236         dict_info["Matriz_C"] = matriz_C.tolist()
237         dict_info["Dimensiones_Matriz_C"] = matriz_C.shape
238 
239         # Se muestra el diccionario de información.
240         logger.info(dict_info)
241         # Se imprimen los tiempos de ejecución.
242         dict_output = {}
243         dict_output[f"TE_Hilo_1"] = tiempo_ejecucion_total
```

```
244
245         dict_str_output = '{'
246         dict_str_output += '''"TE_Hilo_1":'''
247         dict_str_output += f'''"{tiempo_ejecucion_total}"'''
248         for i, tiempo in enumerate(tiempos_ejecucion[1:]):
249             dict_str_output += f''',"TE_Hilo_{i+2}":'''
250             dict_str_output += f'''"{tiempo}"'''
251         dict_str_output += '}'
252 
253         print(dict_str_output)
254     else:
255         time.sleep(0.005)
256         # Procede a calcular la matriz resultante.
257         matriz_C = np.dot(matriz_A, matriz_B)
258         # Se obtiene el tiempo de ejecución.
259         tiempo_ejecucion = time.time() - tiempo_inicio
260         msg = f"Tiempo de ejecución: {tiempo_ejecucion} segundos."
261         dict_info["TE_Hilo_1"] = msg
262         # Se muestra la matriz resultante.
263         dict_info["Matriz_C"] = matriz_C.tolist()
264         dict_info["Dimensiones_Matriz_C"] = matriz_C.shape
265         logger.info(dict_info)
266         dict_str_output = '{'
267         dict_str_output += '''"TE_Hilo_1":'''
268         dict_str_output += f'''"{tiempo_ejecucion}"'''
269         dict_str_output += '}'
270         print(dict_str_output)
271     sys.exit(0)
```

Cuadro 41: Código para pruebas de rendimiento de la Raspberry Pi.

14.2. Código para generación de pruebas de rendimiento

```
1  import subprocess       # Para ejecutar archivos python desde la terminal.
2  import logging          # Para crear logs de los resultados de las pruebas.
3  import json             # Para leer y escribir archivos json.
4  from pythonjsonlogger import jsonlogger # Para crear logs en formato json
5 
6  # Se definen las rutas de los archivos de código.
7  UART_MASTER_CODE = "/clusterfs/docs/protocols/uart_code/uart_master.py"
8  I2C_MASTER_CODE  = "/clusterfs/docs/protocols/i2c_code/I2C_master.py"
9  RPI_CODE         = "/clusterfs/docs/protocols/rpi/rpi_mult.py"
10 ETHERNET_CODE    = "/clusterfs/docs/protocols/ethernet/Ethernet_mpi.py"
11 # Se definen las rutas de los archivos de log.
12 UART_MASTER_LOG  = "/clusterfs/docs/logs/uart/UART_master_time.log"
13 I2C_MASTER_LOG   = "/clusterfs/docs/logs/i2c/I2C_master_time.log"
14 RPI_LOG          = "/clusterfs/docs/logs/rpi/rpi_time.log"
15 # Se definen las rutas de los archivos json.
16 I2C_JSON         = "/clusterfs/docs/logs/i2c/i2c_master_time.json"
17 UART_JSON        = "/clusterfs/docs/logs/uart/uart_master_time.json"
18 RPI_JSON         = "/clusterfs/docs/logs/rpi/rpi_time.json"
19 # Se definen los tamaños de las matrices a probar (2x2, 3x3, ..., 7x7).
20 MATRIX_SIZE      = [f"{i}x{i}" for i in range(2, 8)]
21 # Se definen los esclavos a probar (para protocolos I2C y UART).
22 SLAVES           = ["0x47", "0x47,0x48"]
23 # Se definen la cantidad de repeticiones de cada prueba.
24 REPETICIONES     = 10
25 
26 # Función para ejecutar un archivo python desde la terminal.
27 def escribir_archivo_json(ruta_archivo="", datos={}, log=logging.getLogger()):
28     """
```

```
29      Escribe un diccionario en un archivo json.
30
31      :param ruta_archivo: Ruta del archivo a escribir.
32      :param datos: Diccionario con los datos a escribir.
33      """
34      # Se verifica que la ruta del archivo no esté vacía.
35      if ruta_archivo == "":
36          log.error("La ruta del archivo no puede estar vacía.")
37          return
38      # Se verifica que los argumentos sean una lista de strings.
39      if type(datos) != dict:
40          log.error("Los datos deben ser un diccionario.")
41          return
42      # se recupera la información anterior del archivo, si existe.
43      datos_anteriores = {}
44      try:
45          with open(ruta_archivo, 'r') as archivo:
46              datos_anteriores = json.load(archivo)
47      except:
48          pass
49      # Se agregan los datos nuevos a los datos anteriores.
50      datos = {**datos_anteriores, **datos}
51      # Se escribe el archivo.
52      with open(ruta_archivo, 'w') as archivo:
53          json.dump(datos, archivo, indent=4)
54      return # Se retorna la salida del archivo.
55  def ejecutar_archivo_python(ruta_archivo="", argumentos=[],
56                              log=logging.getLogger()):
57      """
58      Ejecuta un archivo python desde la terminal.
59
60      :param ruta_archivo: Ruta del archivo a ejecutar.
61      :param argumentos: Lista de argumentos a pasar al archivo.
62      """
63      # Se verifica que la ruta del archivo no esté vacía.
64      if ruta_archivo == "":
65          log.error("La ruta del archivo no puede estar vacía.")
66          return
67      # Se verifica que los argumentos sean una lista de strings.
68      if type(argumentos) != list:
69          log.error("Los argumentos deben ser una lista de strings.")
70          return
71      #! Agregar el sudo para ejecutar en la raspberry.
72      lista_parametros = ["python", ruta_archivo] + argumentos
73      # Se inicializa la variable que almacenará la tarea completada.
74      tarea_completada = subprocess.CompletedProcess[str]
75      salida = "" # Se inicializa la variable que almacenará la salida.
76      try:
77          # Se ejecuta el archivo.
78          tarea_completada = subprocess.run(lista_parametros, text=True,
79                                            capture_output=True)
80      except Exception as e:
81          # Se muestra el error.
82          log.error("Error al ejecutar el archivo: " + ruta_archivo)
83          log.error(e)
84      # Se verifica que la tarea se haya completado.
85      if tarea_completada.returncode != 0:
86              log.error("Error al ejecutar el archivo: " + ruta_archivo)
87              log.error(tarea_completada.stderr)
88      # Se recupera la salida del archivo.
89      salida = tarea_completada.stdout
90      return salida # Se retorna la salida del archivo.
91  def configurar_logger(nombre, archivo_log):
92      # Se crea el logger con el nombre especificado.
93      logger = logging.getLogger(nombre)
94      # Se configura el nivel del logger.
95      logger.setLevel(logging.INFO)
96      # Se crea el formateador.
```

```
97      formatter = logging.Formatter('%(asctime)s| %(name)s | %(message)s')
98      # Se crea el handler para la salida a un archivo.
99      file_handler = logging.FileHandler(archivo_log)
100     # Se configura el nivel del handler.
101     file_handler.setLevel(logging.INFO)
102     # Se agrega el formateador al handler.
103     file_handler.setFormatter(formatter)
104     # Se agrega el handler al logger.
105     logger.addHandler(file_handler)
106     return logger # Se retorna el logger.
107 def configurar_logger_json(nombre, archivo_log):
108     # Se crea el logger con el nombre especificado.
109     logger = logging.getLogger(nombre)
110     # Se configura el nivel del logger.
111     logger.setLevel(logging.INFO)
112     # Se crea el formateador.
113     formatter = jsonlogger.JsonFormatter(
114         fmt="%(asctime)s %(levelname)s %(message)s",
115         json_encoder=json.JSONEncoder,
116         json_default=str,
117         json_indent=4
118     )
119     # Se crea el handler para la salida a un archivo.
120     file_handler = logging.FileHandler(archivo_log)
121     # Se configura el nivel del handler.
122     file_handler.setLevel(logging.INFO)
123     # Se agrega el formateador al handler.
124     file_handler.setFormatter(formatter)
125     # Se agrega el handler al logger.
126     logger.addHandler(file_handler)
127     return logger # Se retorna el logger.
128 def pruebas_I2C():
129     """ Esta función ejecuta las pruebas para los protocolos I2C.
130     """
131     # Crear loggers con diferentes nombres y archivos de log
132     logger_UART = configurar_logger('Master_UART', UART_MASTER_LOG)
133     logger_I2C = configurar_logger('Master_I2C', I2C_MASTER_LOG)
134     contador = 1
135     for slave in SLAVES:
136         for size in MATRIX_SIZE:
137             for prueba in range(REPETICIONES):
138                 # Se ejecuta el archivo I2C.
139                 salida_I2C  = ejecutar_archivo_python(I2C_MASTER_CODE,
140                                                       [size, size, slave],
141                                                       logger_I2C)
142                 # Se escribe la salida en el log.
143                 log_message = f"Prueba {prueba + contador}_{size}_{slave}"
144                 log_message += f":{salida_I2C}"
145                 logger_I2C.info(log_message)
146     return
147 def pruebas_UART():
148     """ Esta función ejecuta las pruebas para el protocolo UART.
149     """
150     # Crear loggers con diferentes nombres y archivos de log
151     logger_UART = configurar_logger_json('Master_UART', UART_JSON)
152     contador = 1
153     for slave in SLAVES:
154         for size in MATRIX_SIZE:
155             for prueba in range(REPETICIONES):
156                 # Se ejecuta el archivo UART.
157                 salida_UART = ejecutar_archivo_python(UART_MASTER_CODE,
158                                                       [size, size, slave],
159                                                       logger_UART)
160                 # Se escribe la salida en el log.
161                 log_message = f"Prueba {prueba + contador}_{size}_{slave}"
162                 log_message += f":\n{salida_UART}"
163                 logger_UART.info(log_message)
164     return
```

```python
def pruebas_RPI():
    """ Esta función ejecuta las pruebas para la única Raspberry Pi con y sin
    hilos.
    """
    # Crear loggers con diferentes nombres y archivos de log
    logger_RPI = configurar_logger_json('RPI', RPI_LOG)
    contador = 1
    for hilos in range(1, 4):
        for size in MATRIX_SIZE:
            for prueba in range(REPETICIONES):
                # Se ejecuta el archivo RPI.
                salida_RPI = ejecutar_archivo_python(RPI_CODE,
                                                     [size, size, f"-m{hilos}"],
                                                     logger_RPI)
                # Se escribe la salida en el log.
                dict_salida = {}
                dict_salida_info = {}
                dict_salida_info["Prueba"] = f"{prueba+contador}"
                dict_salida_info["Hilos"] = f"{hilos}"
                dict_salida_info["Dimension"] = f"{size}"
                if isinstance(salida_RPI, str):
                    salida_RPI = json.loads(salida_RPI)
                dict_salida_info["Salida"] = salida_RPI
                n=prueba+contador
                dict_salida[f"Prueba_{n}_{size}_H{hilos}"] = dict_salida_info
                escribir_archivo_json('prueba.json', dict_salida, logger_RPI)
    return
def pruebas_Ethernet(nodos=3, tareas_por_nodo=3):
    """ Esta función ejecuta las pruebas del protocolo Ethernet.
    Argumento:
        nodos {int} -- Define la cantidad de nodos a utilizar (default: {3})
        tareas_por_nodo {int} -- Define la cantidad de tareas por nodo
        (default: {3})
    """
    # Se definen los parámetros de sbatch.
    particion = "world"     # Nombre de la partición.
    ruta_archivo = "/clusterfs/docs/protocols/ethernet/pruebas_ethernet.sh"
    # Se genera el contenido del archivo.
    # Se define el apartado que no cambia ni por dimension ni por prueba.
    inicio_contenido = f'''#!/bin/bash\n#SBATCH --nodes={nodos}\n'''
    inicio_contenido += f'''#SBATCH --partition={particion}\n'''
    inicio_contenido += f'''#SBATCH --ntasks-per-node={tareas_por_nodo}\n'''
    inicio_contenido += "cd /clusterfs\n"
    # Se recorre la lista de matrices.
    for dimension in MATRIX_SIZE:
        # Se calcula la cantidad de procesos necesarios.
        procesos = int(dimension.split('x')[0])
        # Se define el apartado que cambia por dimension.
        ejecutable_python = f"mpiexec --allow-run-as-root -n {procesos}"
        ejecutable_python+= f" python3 {ETHERNET_CODE} {dimension} {dimension}"
        # Se generan la cantidad de pruebas necesarias.
        for prueba in range(REPETICIONES):
            # Se concatenan los strings anteriores en contenido y se agrega el
            # parámetro que cambia por prueba.
            contenido = inicio_contenido+ejecutable_python+f" -np{prueba+1} "
            contenido += f"-cn{nodos} -pp{tareas_por_nodo}"
            # Se escribe el contenido en el archivo.
            with open(ruta_archivo, 'w') as archivo:
                archivo.write(contenido)
            mensaje = f"Se ejecuta la prueba {prueba+1} de "
            mensaje += f"la dimensión {dimension}."
            print(mensaje)
            try:
                # Se ejecuta el archivo.
                subprocess.run(["sudo", "sbatch", ruta_archivo])
            except Exception as e:
                # Se muestra el error.
                print("Error al ejecutar el archivo: "+ruta_archivo)
```

```
233                 print(e)
234             except KeyboardInterrupt:
235                 # Se muestra el error.
236                 print("Se interrumpió la ejecución del archivo: "+ruta_archivo)
237                 return
238     return
239 pruebas_Ethernet(3,3)
240 pruebas_I2C()
241 pruebas_RPI()
242 pruebas_UART()
```

Cuadro 42: Código de generación de pruebas de rendimientos.